ARMY RESEARCH LABORATORY

A Terminal Guidance Model for Smart Projectiles Employing a Semi-Active Laser Seeker

by Luke S. Strohm

ARL-TR-5654 **August 2011**

NOTICES

Disclaimers

The findings in this report are not to be construed as an official Department of the Army position unless so designated by other authorized documents.

Citation of manufacturer's or trade names does not constitute an official endorsement or approval of the use thereof.

Destroy this report when it is no longer needed. Do not return it to the originator.

Army Research Laboratory
Aberdeen Proving Ground, MD 21005-5066

ARL-TR-5654 **August 2011**

A Terminal Guidance Model for Smart Projectiles Employing a Semi-Active Laser Seeker

Luke S. Strohm
Weapons and Materials Research Directorate, ARL

REPORT DOCUMENTATION PAGE	Form Approved OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED (From - To)
August 2011	Final	1 January 2010–31 March 2011

4. TITLE AND SUBTITLE
A Terminal Guidance Model for Smart Projectiles Employing a Semi-Active Laser Seeker

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)
Luke S. Strohm

5d. PROJECT NUMBER
AH80

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
U.S. Army Research Laboratory
ATTN: RDRL-WML-A
Aberdeen Proving Ground, MD 21005-5066

8. PERFORMING ORGANIZATION REPORT NUMBER
ARL-TR-5654

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES

14. ABSTRACT
This report describes the development of a semi-active laser terminal guidance model. The C++ class implementation of the model, *sSalSeeker*, can be executed stand-alone or be embedded into a larger guided projectile model. It was validated using the NVLaserD model written by the Night Vision and Electronic Sensors Directorate of the Communications-Electronics Research, Development, and Engineering Center.

The first objective of the model is to determine the laser power distribution at the seeker, accomplished by calculating the laser beam's transmission path and power loss. The beam is modeled stochastically through a set of rays forming a solid cone of the given divergence. Each ray's transmission path and power loss is calculated and its power is summed with the other rays' contributions at the seeker. Ray tracing is used to determine the ray-transmission paths from designator to target and target to seeker. The Beer-Lambert Law is used to compute power loss due to atmospheric attenuation. Lambert's Cosine Law and designator-seeker-target geometry are used to determine the portion of power that the seeker receives after it reflects off of the target. The second objective is to convert the seeker power distribution into projectile guidance signals. Guidance signals are calculated by dividing the power received in the pitching and yawing directions by the total power received by the seeker. The guidance signals are not considered reliable unless the seeker power is above a signal-to-noise threshold.

15. SUBJECT TERMS
semi-active laser, terminal guidance, smart projectile, C++, stochastic model

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Luke S. Strohm
Unclassified	Unclassified	Unclassified	UU	82	19b. TELEPHONE NUMBER (*Include area code*) 410-278-6104

Contents

List of Figures

List of Tables

Acknowledgments

Several colleagues from the U.S. Army Research Laboratory's Weapons and Materials Research Directorate contributed to the work described in this report. Dr. William Oberle and Mr. Richard Pearson provided guidance and inspiration to the project, as well as editorial recommendations to the report. Mr. Robert Yager offered key insights about the physics of laser transmissions, and his advice on C++ programming was very helpful. Dr. Chase Munson provided expertise on beam profiles and divergence. Mr. Andrew Thompson made connections with outside laboratories, particularly the Communications-Electronics Research, Development, and Engineering Center's Night Vision and Electronic Sensors Directorate, whose model was used to validate ours.

INTENTIONALLY LEFT BLANK.

1. Introduction

This report describes the development of a semi-active laser (SAL) terminal guidance model. SAL guidance typically consists of a scout illuminating a target with short, high-energy laser pulses in a near-infrared (IR) wavelength (figure 1). Some of the energy from each pulse reflects off of the target and into the seeker's multi-section IR detector. By comparing the power measured in each section of the IR detector, the seeker approximates the location of the reflected power within its field of view (FOV).* Using this information, the seeker returns control signals to the projectile's maneuver system to steer towards the target.

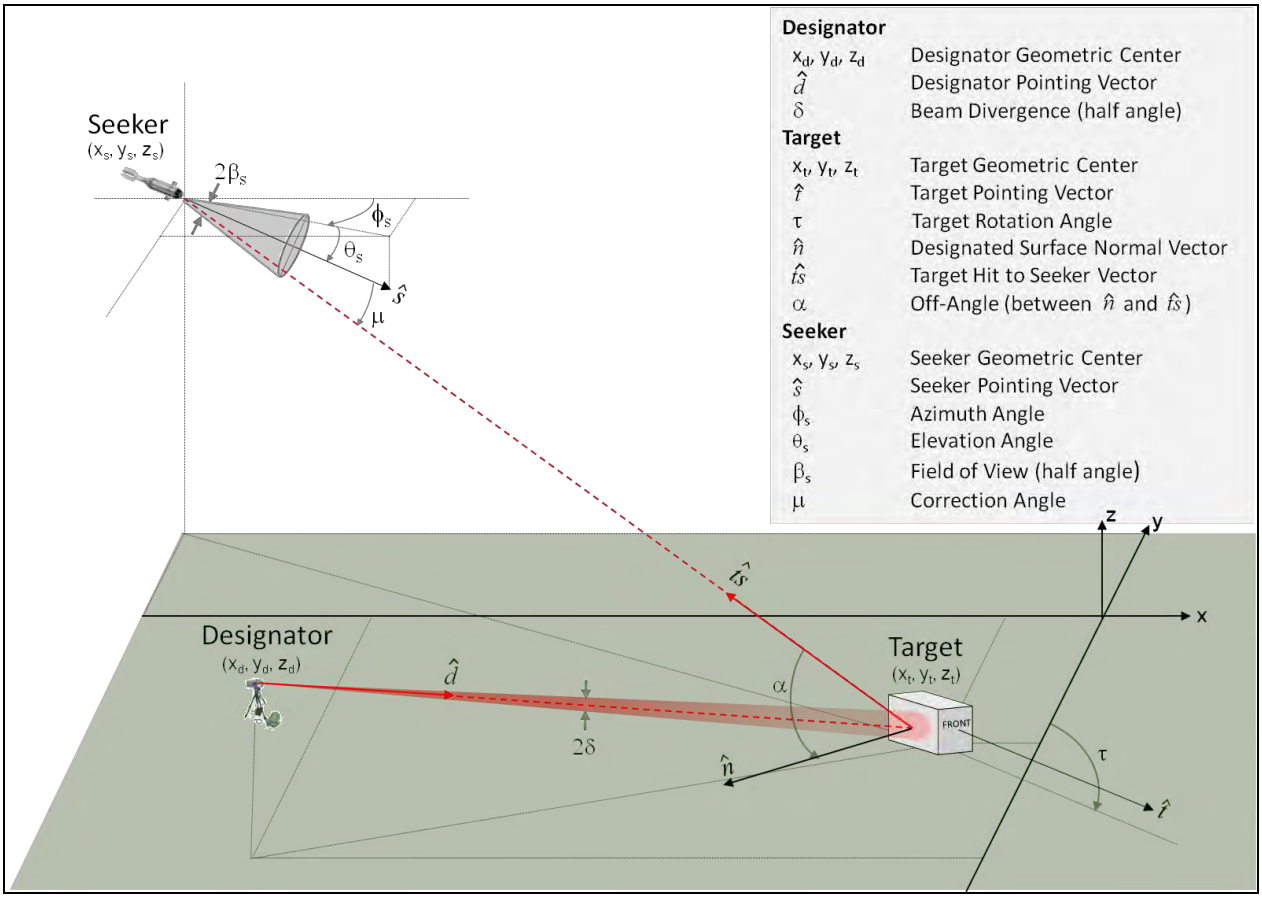


Figure 1. SAL terminal guidance.

*In this model, the seeker is assumed to be inside the nose of the projectile.

The first objective of the model is to determine the laser power distribution at the seeker. This is accomplished by calculating the laser beam's transmission path and power loss. The beam is modeled stochastically through a set of rays that form a solid cone of the given divergence. Each ray's transmission path and power loss is calculated, and its power is summed with the other rays' contributions at the seeker. Ray tracing determines ray transmission paths from designator to target, and target to seeker. The Beer-Lambert Law is used to compute power loss due to atmospheric attenuation. Lambert's Cosine Law and designator-seeker-target geometry are used to determine the portion of the power that the seeker receives after it reflects off of the target.

The second objective of the model is to convert the seeker power distribution into projectile guidance signals. The guidance signals are calculated by dividing the power received in the pitching and yawing directions by the total power received by the seeker. The guidance signals are not considered reliable unless the seeker power is above a signal-to-noise (S/N) threshold.

The following are the model's primary simplifying assumptions:

- Flat Earth, Flat Terrain: At ranges typical for SAL guidance (<25 km), the curvature of the earth does not significantly affect the model geometry. Flat terrain greatly simplified the model development, although it is listed as a potential area for further development in section 7 of this report (Path Forward).
- Linear Optics: Because semi-active designator lasers are typically of relatively low power, nonlinear optical effects (e.g., thermal blooming) are not included in the model.
- Simplified Atmospheric Transmission: Atmospheric attenuation through absorption and reflection is incorporated only as a gross power loss to the transmission. It is assumed that the power reflected or reemitted from the transmission path to the seeker is negligible (i.e. phenomena such as backscatter are not modeled).
- Lambertian Reflection: Lambertian reflection assumes perfectly diffuse (matte) reflective surfaces. This is another suggested area for further development discussed in section 7.

The C++ class implementation of the model, *sSalSeeker*, can be executed stand-alone or be embedded into a larger guided projectile model. The model was validated using the Night Vision Laser Designator (NVLaserD) model written by the Night Vision and Electronic Sensors Directorate (NVESD) of the Communications-Electronics Research, Development, and Engineering Center (CERDEC).

The report is organized into the following sections: Section 2 describes the model coordinate systems and major entities (bodies). Section 3 explains the underlying physics of the laser transmission algorithm. Section 4 describes the seeker guidance algorithm. Section 5 discusses the model's C++ class implementation. Section 6 gives the results of the model's validation, and we conclude in section 7 with some recommendations for the use and further development of the model.

2. Model Setup

2.1 Global Coordinate System

The global coordinate system is Cartesian and fixed to the earth's surface, which is assumed to be an infinite plane. The global coordinate system is defined by three orthogonal unit vectors: $\hat{x}, \hat{y}$, and $\hat{z}$.* The x-y plane forms the earth's surface (ground plane), where z = 0. As z increases, the height above the earth's surface increases. Figure 1 showed the model set-up in global coordinates.

The orientation of the x and y axes and the coordinate system origin are defined by the user with the following conditions:

1. The x and y axes may be oriented in any direction on the ground plane, provided they follow the right-hand rule ($\hat{x} \times \hat{y} = \hat{z}$).
2. The origin of the global coordinate system may be situated anywhere on the ground plane.

The global coordinate system is the default system used throughout the model. If not specified, one should assume these coordinates. A major purpose of the global coordinate system is to define the relative positions and orientations of the model's bodies, which is described in the next section.

2.2 Bodies (Designator, Target, Seeker)

The bodies involved in SAL guidance are the designator, target, and seeker. The user is responsible to input all body positions and orientations, with the exception of designator orientation. From the orientations, the model calculates normalized pointing vectors (unit vectors) for each body.

2.2.1 Designator

The designator is assumed to be a point-mass, whose coordinates are inputs to the model. The designator pointing vector, $\hat{d}$, points from the designator location toward the target's geometric center.

2.2.2 Target

The target is a rectangular parallelepiped, with dimensions of length, width, and height (figure 2). The target is assumed to always stay upright, meaning its top surface stays parallel to the ground plane. The target pointing vector, $\hat{t}$, points from the target's geometric center toward the center of the target's front surface (shown in blue in the figure). The target's right and left sides

*All unit vectors will be denoted by the ^ symbol.

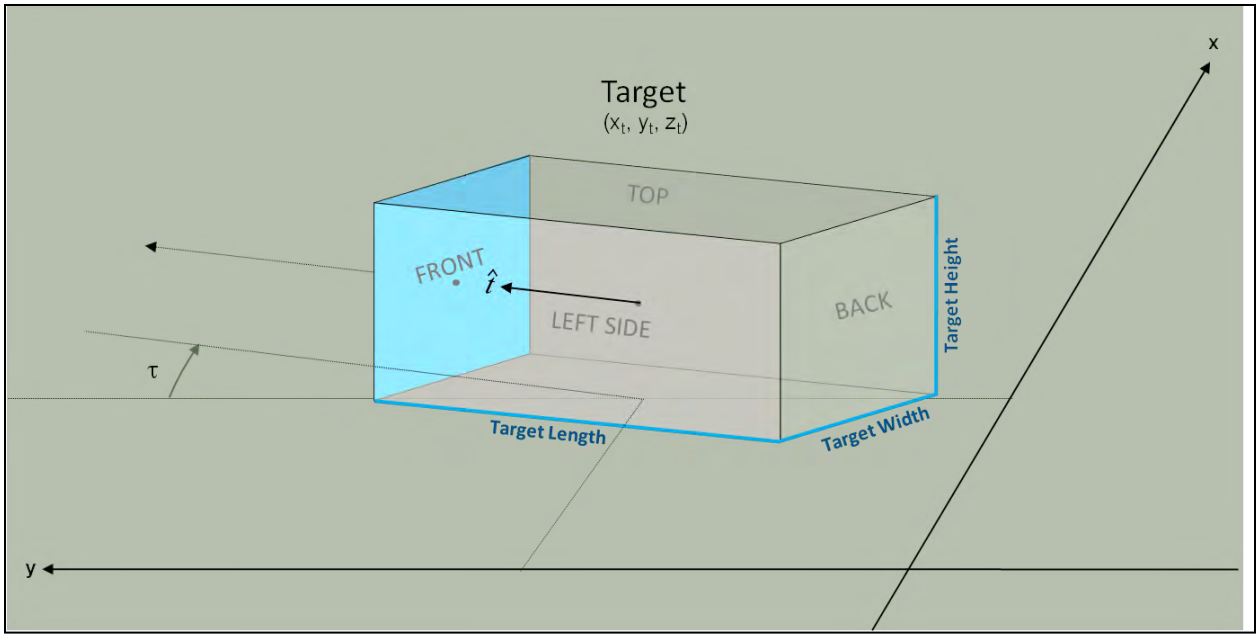


Figure 2. Rectangular parallelepiped target.

are defined by the respective right- and left-handed directions when facing from the target's geometric center towards the front of the target.

The orientation of the target is specified by the user through a target rotation angle, τ, which is the angle between the global y axis and the projection of the vector pointing away from the target's front surface onto the x-y plane measured clockwise as seen from above (*Bounds: 0 to* 2π).

2.2.3 Seeker

The model's SAL seeker is modeled as a four-quadrant IR detector mounted in the nose of a projectile. It is modeled as a zero-thickness circular disc. The seeker pointing vector, $\hat{s}$, aligns with the long axis of the projectile and points from the center of the seeker normal to the seeker's plane and away from the projectile. The seeker's FOV is the angular measure of the cone centered about $\hat{s}$ in which the seeker receives radiation.

The user specifies $\hat{s}$ through azimuth and elevation angles:

- Azimuth (ϕ) – the angle between the global x axis and the projection of $\hat{s}$ onto the x-y plane. Azimuth is positive in the counter-clockwise direction as seen from above. *Bounds:* $-\pi$ *to* π.
- Elevation (θ) – the angle between the global x-y plane and $\hat{s}$. Elevation is positive when $\hat{s}$ points above the x-y plane. *Bounds:* $-\pi/2$ *to* $\pi/2$.

The seeker is divided by yaw and pitch axes, which will be discussed in the next section.

2.3 Body-Fixed Coordinate System

A Cartesian body-fixed coordinate system (x′, y′, z′) is used to define vectors relative to a body pointing vector, $\hat{b}$ (section 2.2). In this ~~–~~prime" coordinate system used by Yager (*1*), the origin is set at the geometric center of the local body (equation 1). The x′ axis is aligned with the body pointing vector, $\hat{b}$. The y′ axis points orthogonal to the left of $\hat{b}$ and parallel to the global x-y plane. The z′ axis is formed orthogonal to the x′ and y′ axes according to the right-hand rule (figure 3).

In equation form:

$$\hat{x}' = \hat{b}. \tag{1}$$

$$\hat{y}' = \hat{z} \times \hat{b}. \tag{2}$$

$$\hat{z}' = \hat{x}' \times \hat{y}'. \tag{3}$$

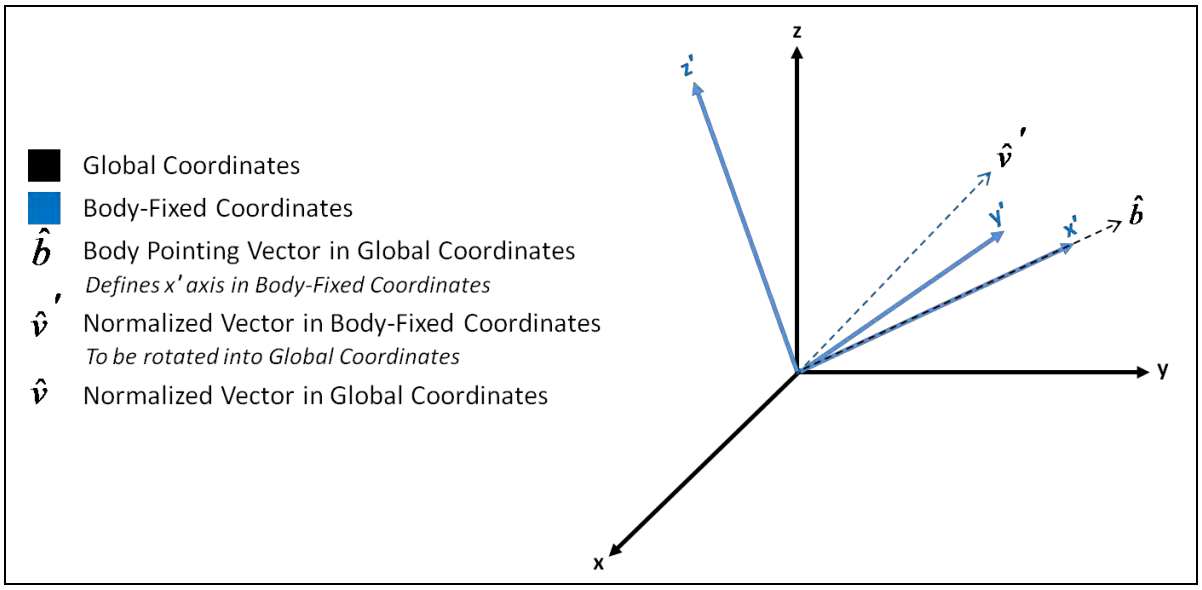


Figure 3. Body-fixed coordinates.

In the prime system, a vector, $\hat{v}'$, is defined. The methodologies to define $\hat{v}'$ are discussed in sections 2.3.1–2.3.3 of this report. To transform $\hat{v}'$ from the body-fixed system to the global coordinate system, we translate to the location of the body in the global system and then rotate to align the x′ axis with the body's pointing vector, $\hat{b}$. To accomplish this rotation, the model uses a rotation matrix that is derived by Yager (*1*).* To find the vector in global coordinates ($\hat{v}$), the rotation matrix, A, is multiplied by $\hat{v}'$:

*The rotation matrix is shown in a simplified form used for unit vectors.

$$\hat{v} = A\hat{v}', \tag{4}$$

where

$$\mathrm{A} = \begin{bmatrix} b_x & -\gamma b_y & -\gamma b_x b_z \\ b_y & -\gamma b_x & -\gamma b_y b_z \\ b_z & 0 & \frac{1}{\gamma} \end{bmatrix}, \tag{5}$$

where b_x , b_y , and b_z are the components of the body pointing vector, $\hat{b}$, in global coordinates and:

$$\gamma = \frac{1}{\sqrt{1-{b_z}^2}}. \tag{6}$$

2.3.1 Designator Application

For the designator, body-fixed coordinates are used for two purposes:

1. Adjusting the designator pointing vector ($\hat{d}$) for aim error. This new pointing vector is called $\hat{d}_{\mathrm{aim}}$.
2. Modeling the beam's solid divergence cone that is centered on $\hat{d}_{\mathrm{aim}}$. This is done stochastically by dividing the beam into a set of rays. Body-fixed coordinates are used to calculate each individual ray vector, $\hat{r}_{\mathrm{ind}}$.

2.3.1.1 Aim Error. To model aim error, $\hat{d}$ is perturbed through horizontal and vertical perturbation angles, $\alpha_{y'}$ and $\alpha_{z'}$.

Referring to figure 4, the perturbed vector is expressed in the body-fixed coordinate system by adding the perturbations to the original pointing vector.

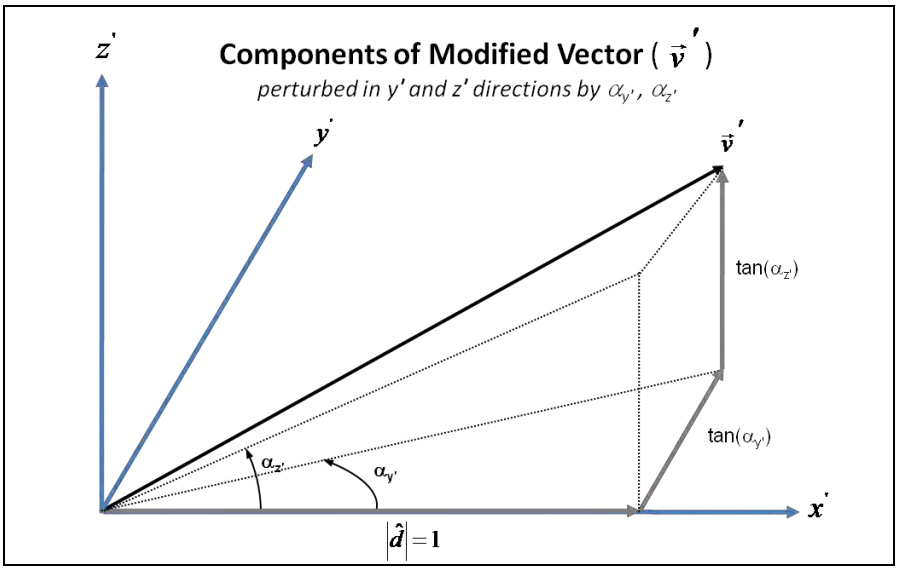


Figure 4. Modified vector in body-fixed coordinates.

$$\vec{v}' = 1 \cdot \hat{x}' + tan(\alpha_{y'}) \cdot \hat{y}' + tan(\alpha_{z'}) \cdot \hat{z}'. \quad (7)$$

To normalize the perturbed vector, $\vec{v}'$, it is divided by its magnitude:

$$|\vec{v}'| = \sqrt{1 + (tan(\alpha_{y'}))^2 + (tan(\alpha_{z'}))^2}. \quad (8)$$

$$\hat{v}' = \frac{\vec{v}'}{|\vec{v}'|}. \quad (9)$$

Finally, we transform $\hat{v}'$ into the global coordinate system using equation 4. This results in our new designator pointing vector, $\hat{d}_{\text{aim}}$.

2.3.1.2 Solid Beam Divergence Cone**.** To calculate the direction of each ray ($\hat{r}_{\text{ind}}$) in the beam's solid divergence cone, we define a perturbed vector, $\hat{v}'$, in the same way for aim error (equations 7–9, figure 5). The beam is assumed to be circular, so that $\alpha_{y'}$ and $\alpha_{z'}$ vary over the same range. The selection of $\alpha_{y'}$ and $\alpha_{z'}$ is done through random draws from a normal distribution, which is discussed in section 3.1.1.

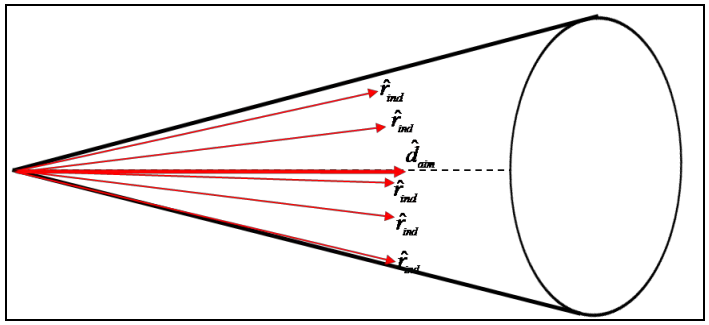


Figure 5. Calculation of individual ray vectors.

2.3.2 Target Application

Because of the target's simplified geometry and rotations, the model does not currently use body-fixed coordinates for the target. However, if the model were updated to include complex target geometry or rotations, body-fixed coordinates could greatly simplify the characterization of target surfaces.

2.3.3 Seeker Application

For the seeker, body-fixed coordinates are used to find the yaw and pitch axes in the seeker plane. In the body-fixed coordinate system, the pitch axis is the y′ axis, and the yaw axis is the z′ axis (figure 6).

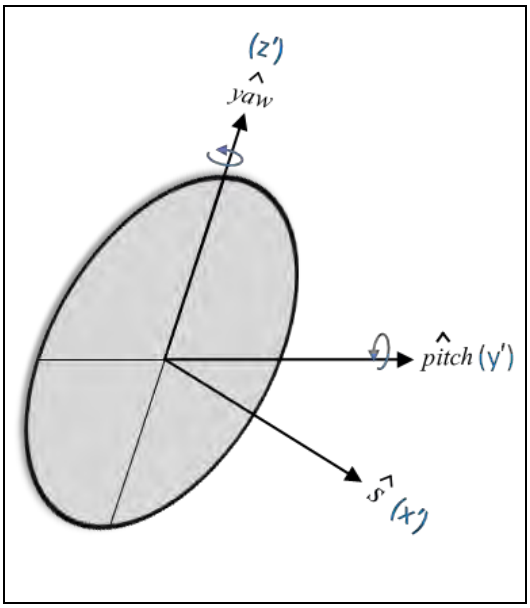


Figure 6. Pitch and yaw axes of the seeker plane.

Thus, to find the pitch and yaw axes in global coordinates, we first set the vector $\hat{v}'$ equal to each axis in body-fixed coordinates:

$$\hat{v}_{pitch}{}' = \mathrm{y}' = (0,1,0). \tag{10}$$

$$\hat{v}_{yaw}{}' = \mathrm{z}' = (0,0,1). \tag{11}$$

After rotating back to global coordinates using the rotation matrix (A), we have the pitch and yaw axes in global coordinates.

3. Laser Transmission Model

The laser transmission algorithm follows four successive stages (figure 7). In each stage, the model calculates the beam's transmission path and power loss:

- Stage 1: Atmospheric Transmission (Designator to Target)
- Stage 2: Target Reflection
- Stage 3: Atmospheric Transmission (Target to Seeker)
- Stage 4: Seeker Reception

} *Combined into one stage in this report*

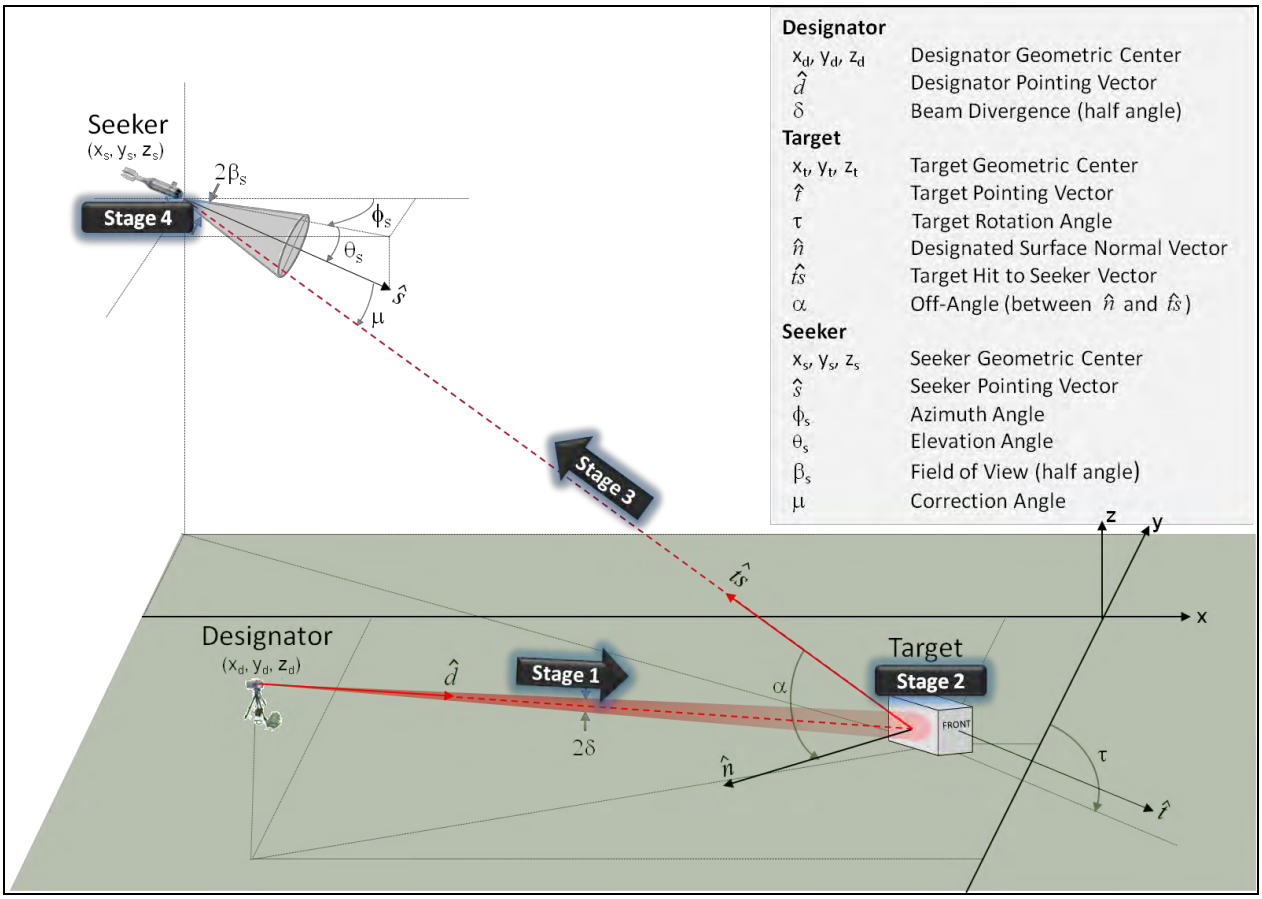


Figure 7. Laser transmission stages.

3.1 Stage 1: Atmospheric Transmission

3.1.1 Beam Divergence

Prior to modeling the beam divergence, the beam's centerline must first be defined. The designator is assumed to always aim for the geometric center of the target, and this pointing vector is perturbed in random horizontal and vertical directions to account for designator error, as described in section 2.3.1. The perturbation angles are drawn from normal distributions with standard deviations σ_{α_y}, and σ_{α_z}. This results in the vector $\hat{d}_{\text{aim}}$.

As the beam emerges from the designator, it diverges along its transmission path. Assuming the beam to have a Gaussian profile across its transverse axis, the divergence, δ, is commonly defined to be the half angle corresponding to the location along the transverse axis where the intensity* drops to $1/e^2$ times the intensity at the beam centerline (figure 8) (*2*). At any range from the designator, the diameter of the cone swept out at this point on the transverse axis is

*Intensity is power per unit solid angle, and it will be discussed in section 3.1.2.

referred to as the beam diameter.* This occurs at two standard deviations from the beam centerline intensity, meaning that ~95% of the laser beam's total power is within the cone of the given divergence angle.

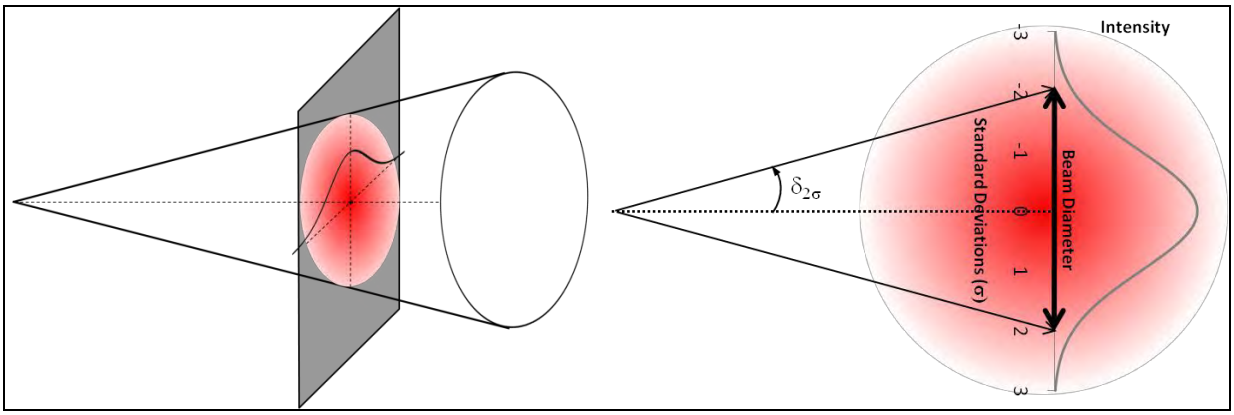


Figure 8. Gaussian beam with $1/e^2$ beam diameter.

Next, the beam is divided stochastically into a set of rays, with each ray carrying its individual portion of the total power. Each ray's direction ($\hat{r}_{\text{ind}}$) is determined using body-fixed coordinates, as described in section 2.3.1. The perturbation angles, $\alpha_{y'}$ and $\alpha_{z'}$, are drawn from a normal distribution with a standard deviation based on the beam divergence angle ($\sigma_{\alpha_{y',z'}} = \sigma_\delta$).

Recalling that the beam diameter corresponding to δ is defined by $2\sigma_{\text{transverse}}$, which varies closely to $2\sigma_\delta$, we solve for σ_δ:†

$$\delta \cong 2\sigma_\delta. \tag{12}$$

$$\sigma_\delta \cong \frac{\delta}{2}. \tag{13}$$

3.1.2 Attenuation

The first part of atmospheric transmission was determining the direction of the beam, which was accomplished through a division into a set of rays, each with its own direction, $\hat{r}_{\text{ind}}$. The second part of atmospheric transmission is determining the power loss for each ray on its path to the target. The starting power for each ray (ϕ_{ray}) is determined by dividing the beam's pulsed power by the total number of rays (n), and multiplying by a designator efficiency coefficient ($d_{\text{efficiency}}$):

*There are alternative methods for defining beam divergence. It is sometimes defined as the full cone angle of the beam, instead of the half angle that we are assuming. In addition, the beam width is sometimes defined according to the full-width at half maximum method (FWHM).

†This is a small angle approximation, as the divergence by definition produces a normal distribution in distance across the beam's transverse axis ($\sigma_{\text{transverse}}$), and not in the beam's divergence angle. The tangent of the divergence angles, α_y and α_z, is linked to the transverse distance. For small angles, $\tan\alpha \cong \alpha$, and the divergence angles approximately follow a normal distribution.

$$\phi_{ray} = \frac{\phi_{pulse}}{n} \cdot d_{efficiency}. \quad (14)$$

Atmospheric attenuation is the exponential decrease in beam intensity as it transmits through the atmosphere. The Beer-Lambert Law characterizes this attenuation:

$$I = I_0 e^{-kx}, \quad (15)$$

where:

I = Attenuated Intensity at distance x,

I_0 = Initial Intensity,

k = Attenuation Coefficient, and

x = Path Length.

Beam intensity, also known as radiant intensity, is defined as the power (ϕ) per unit solid angle (Ω) subtended by the beam. The solid angle is the surface area subtended on a sphere of radius r (figure 9). It is measured in steradians (sr), where the sphere represents 4π sr.

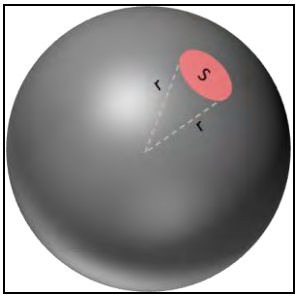


Figure 9. Solid angle.

Thus, the solid angle is calculated by dividing the subtended surface area (S) by the square of the sphere's radius (r):

$$\Omega = \frac{S}{r^2}. \quad (16)$$

In the case of a laser beam, r is the range traversed by the beam measured along its centerline. Because the model does not consider nonlinear optical phenomena, the beam divergence remains constant throughout the transmission. This means that the solid angle also remains constant, and therefore, in this case, we can generalize the Beer-Lambert Law for power (see figure 10 and equations 17-20).

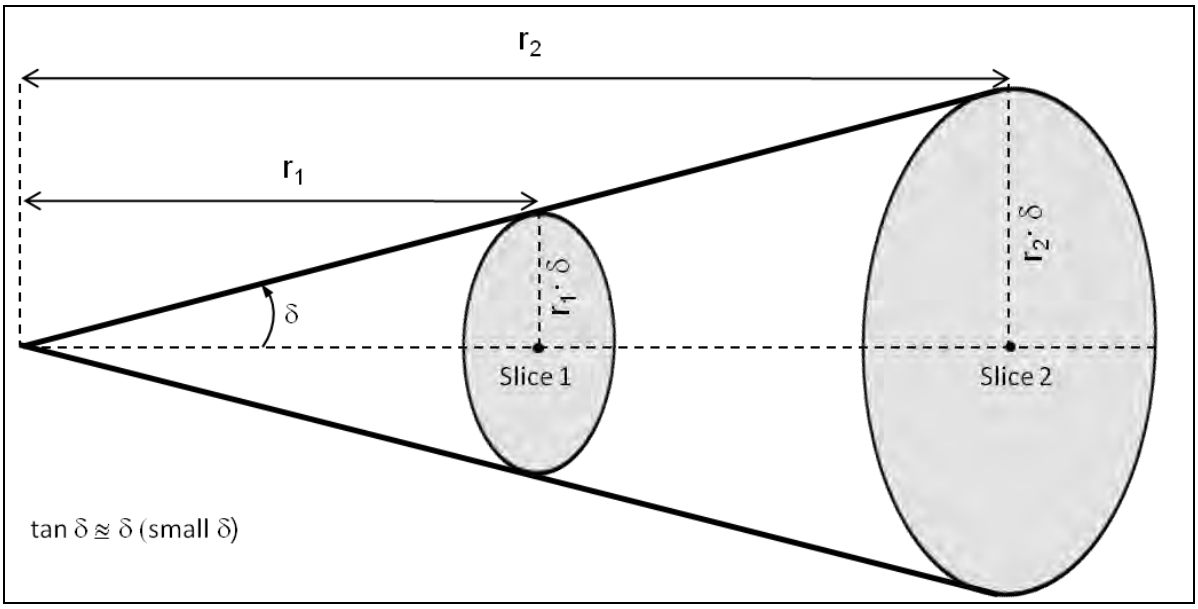


Figure 10. Solid angle calculation at different sections of beam.

For small δ,

$$S_1 \cong \pi \cdot (r_1 \cdot \delta)^2,\ S_2 \cong \pi \cdot (r_2 \cdot \delta)^2. \tag{17}$$

$$\Omega_1 = \frac{S_1}{{r_1}^2} = \frac{\pi \cdot (r_1 \cdot \delta)^2}{{r_1}^2} = \pi \cdot \delta^2,\ \Omega_2 = \frac{S_2}{{r_2}^2} = \frac{\pi \cdot (r_2 \cdot \delta)^2}{{r_2}^2} = \pi \cdot \delta^2. \tag{18}$$

Thus,

$$\Omega_1 = \Omega_2\ . \tag{19}$$

Therefore, for a beam of constant divergence, we can generalize the Beer-Lambert Law for beam power, ϕ:

$$I = \frac{\phi}{\Omega} = \frac{\phi_0}{\Omega_0} e^{-kx},$$

$$\phi = \phi_0 e^{-kx}. \tag{20}$$

Because of the complexity and variability of the earth's atmosphere, the attenuation coefficient (k) varies according to many factors, and it needs to be calculated for each scenario. Equation 21 approximates the attenuation coefficient by summing four major components:

$$k = k_{\text{Molecular Absorption}} + k_{\text{Molecular Scattering}} + k_{\text{Aerosol Absorption}} + k_{\text{Aerosol Scattering}}. \tag{21}$$

Molecular, in the context of equation 21, refers to atmospheric particles larger than electrons but smaller than λ, the laser wavelength. Similarly, *aerosol* refers to particles that have a size comparable to λ (*2*). Given these four attenuating components, the model utilizes lookup tables to determine the attenuation coefficient (*3*). The tables break up the atmosphere into one-kilometer-deep altitude segments. The look-up tables require the following information:

- **Start Height, End Height, Path Length** – if the path traverses multiple altitudes, the model breaks up the path and steps through each altitude segment

- **Laser Type** – characterized by the wavelength. The look-up tables currently handle two laser types:
  Nd:YAG laser (1.06* μm) – the most popular wavelength currently for range-finding and designation, non-eye-safe.
  Er:Glass laser (1.54 μm) – an eye-safe wavelength of interest.

- **Visibility** – affecting aerosol absorption and scattering
  Clear: 23 km
  Hazy: 5 km

- **Latitude** – affecting molecular absorption and scattering
  Tropics: 0° to 23.5° and 0° to –23.5°
  Mid-Latitudes: 23.5° to 50° and –23.5° to –50°
  Sub-Arctic: 50° to 70° and –50° to –70°

- **Season** – affecting molecular absorption and scattering
  Summer: March 22 – September 21
  Winter: September 22 – March 21

Thus, using equations 20 and 21, the model determines the attenuated power of the ray where it intersects the target surface (ϕ_{target}). The distance from the designator to surface intersection point in equation 20 (x), is determined by ray tracing in section 3.2.1.

3.2 Stage 2: Target Reflection

3.2.1 Ray Projection onto Target

The first element of target reflection is determining if and where each ray hits the target. To this end, we employ ray tracing, which determines the first plane of interest that is intersected by the ray and the intersection point on that plane. Wikipedia describes the intersection in matrix notation (*4*):

$$\begin{bmatrix} t \\ u \\ v \end{bmatrix} = \begin{bmatrix} x_a - x_b & x_1 - x_0 & x_2 - x_0 \\ y_a - y_b & y_1 - y_0 & y_2 - y_0 \\ z_a - z_b & z_1 - z_0 & z_2 - z_0 \end{bmatrix}^{-1} \begin{bmatrix} x_a - x_0 \\ y_a - y_0 \\ z_a - z_0 \end{bmatrix}, \tag{22}$$

*Nd:Glass, a laser of nearly identical wavelength to Nd:YAG, was substituted in the attenuation lookup tables.

where

P_o (x_0, y_0, z_0) = reference point on plane,

P_1 (x_1, y_1, z_1) = second point on plane (defining first direction in plane relative to P_o),

P_2 (x_2, y_2, z_2) = third point on plane (defining second direction in plane relative to P_o),

l_a (x_a, y_a, z_a) = starting point of the ray,

l_b (x_b, y_b, z_b) = second point on ray, defining the direction,

t = distance between ray start point and the plane,

u = distance in plane from P_0 to ray intersection in first direction, and

v = distance in plane from P_0 to ray intersection in second direction.

In the model, these elements are defined in global coordinates (figure 11):

Input

P_o = center point of surface.

P_1 = center point of left edge of the surface.

P_2 = center point of top edge of the surface.

l_a = designator location.

l_b - l_a = individual ray vector from designator ($\hat{r}_{ind}$).

Output

t = distance between designator and surface intersected.

u = distance from center of surface to ray intersection in $\hat{u}$ direction.

v = distance from center of surface to ray intersection in $\hat{v}$ direction.

l_b = ray hit point on plane = $P_0 + u\hat{u} + v\hat{v}$.

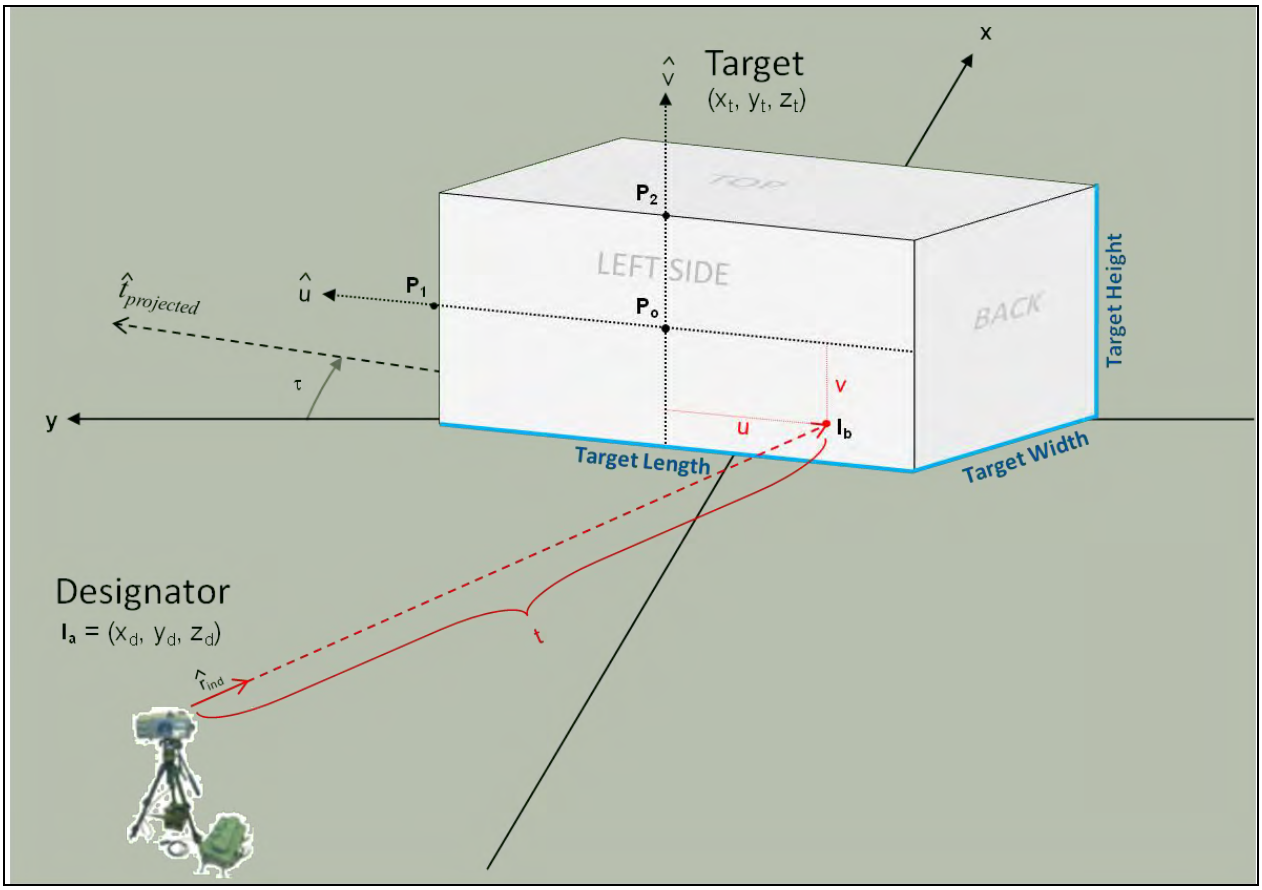


Figure 11. Ray tracing (target-centric x-y plane).

To see if the ray intersects a finite surface, u and v are compared to the dimensions of the surface. For the parallelepiped target, this is half of the target length, width, or height, depending on the surface. If u and v are both less than one half of the dimension, there exists an intersection, unless the ray intersects another surface first. Thus, the model implements the ray-tracing routine for all seven surfaces, checking to see what surface(s) the plane intersects, and, of those, choosing the one with the shortest ray length, t. The first five surfaces tested form a rectangular parallelepiped target: Left Side, Front, Right Side, Back, and Top.* The sixth surface is the ground plane. Ray intersection with the ground plane indicates underspill or overspill. The final surface that the ray-tracing routine checks is the seeker itself. This tests the rare situation in which the laser ray is pointing directly into the seeker from the designator.

*The target's bottom surface is not tested in the ray tracing routine, although it could be added to the model in the future. This capability could be useful if the target was in the air, and the projectile/seeker had the capability to fly upwards towards the target.

3.2.2 Surface Reflection

Once the ray hits a surface, it will either be absorbed or reflected. To model the power lost at the surface due to absorption, the model employs a simple "target reflectivity" multiplier ($t_{reflectivity}$), which can be varied for each surface:

$$\phi_{target,reflected} = \phi_{target,received} \cdot t_{reflectivity}. \quad (23)$$

Several studies have investigated the reflective properties of different targets, which are a result of both surface composition and laser wavelength (*5*).

There are two primary types of reflection (figure 12):

1. Specular Reflection – mirror-like reflection, where the angle of incidence (θ_i) equals the angle of reflection (θ_r).
2. Lambertian Reflection – diffuse reflection, where the reflected energy is scattered in all directions regardless of angle of incidence.

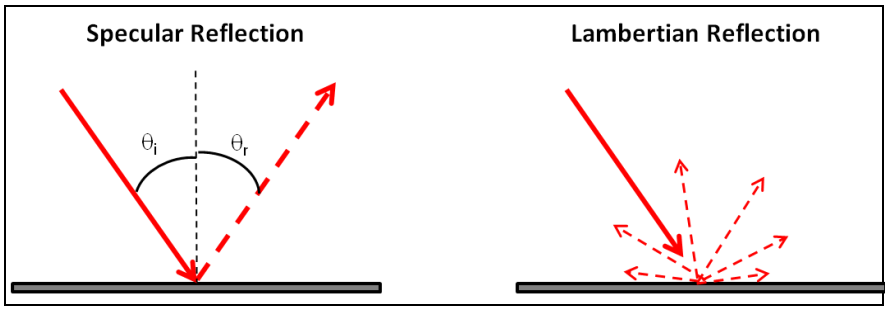


Figure 12. Specular vs. Lambertian reflection.

Real-world scenarios generally have elements of both types of reflection, but Lambertian reflection is closer to actual behavior, and it is used exclusively in the model. Thus, the model assumes that the energy reflects off of the target in all directions in a 180° hemisphere normal to the surface of reflection. This will be described later in section 3.3.2.

3.3 Stages 3 and 4: Atmospheric Transmission and Seeker Reception

3.3.1 Ray Projection Into Seeker

After target reflection, the model has calculated the following data for each ray:

- Coordinates of ray surface intersection (l_b)
- Surface the ray intersects (Left Side, Front, Right Side, Back, Top, Ground,* Direct-to-Seeker, or no surface hit)

*If the ray hits the ground, the model tests whether the ray is obscured from the seeker by the target.

- Ray Power ($\phi_{target,reflected}$)

Using these data, we can now find if and where the rays project into the seeker's four-quadrant, IR sensor. To find if a ray projects into the seeker, it must pass two tests (refer to figure 1):

1. Correction Angle (μ) between seeker heading ($\hat{s}$) and the vector from the seeker to the ray hit point ($\widehat{st}$) must be less than or equal to the seeker FOV (β_s).
2. Off-Angle (α) between the surface normal vector of ray hit ($\hat{n}$) and the vector from the ray hit point to the seeker ($\widehat{ts}$) must be less than 90°.

These angles are determined using the dot product of the two vectors:

$$\mu = \cos^{-1}\left(\frac{\hat{s}\cdot\widehat{st}}{|\hat{s}|\cdot|\widehat{st}|}\right) \leq \beta_s \,. \tag{24}$$

$$\alpha = \cos^{-1}\left(\frac{\hat{n}\cdot\widehat{ts}}{|\hat{n}|\cdot|\widehat{ts}|}\right) < \frac{\pi}{2} \,. \tag{25}$$

If the ray does not pass both of these tests, it does not project into the seeker's sensor.

If the ray passes both tests, the model determines which of the four seeker quadrants the ray projects into. To do so, the model again uses ray tracing. In this instance, the ray originating at the target intersection (l_b) is pointed in the negative direction of $\hat{s}$, and it is determined where it intersects the seeker plane (figure 13). The pitch and yaw axes forming the seeker plane are calculated using body-fixed coordinates (section 2.3.3). The ray tracing produces a $\begin{bmatrix} t \\ u \\ v \end{bmatrix}$ output vector that locates the ray's intersection point with the seeker:

t = ray distance to seeker plane.

u = distance from seeker center to ray intersection in the yaw direction.

v = distance from seeker center to ray intersection in the pitch direction.

A seeker intersection in the positive yaw *direction* (+u) indicates that the seeker must rotate positively about the yaw *axis* (to the left in the FOV) to point at l_b (refer to figure 6). The positive pitch direction (+v) is similarly linked to a positive rotation about the pitch axis (to the top in the FOV). Because we already know that the ray projects into the seeker's sensor, we only need to know whether u and v are positive or negative to determine what quadrant the ray projects into.

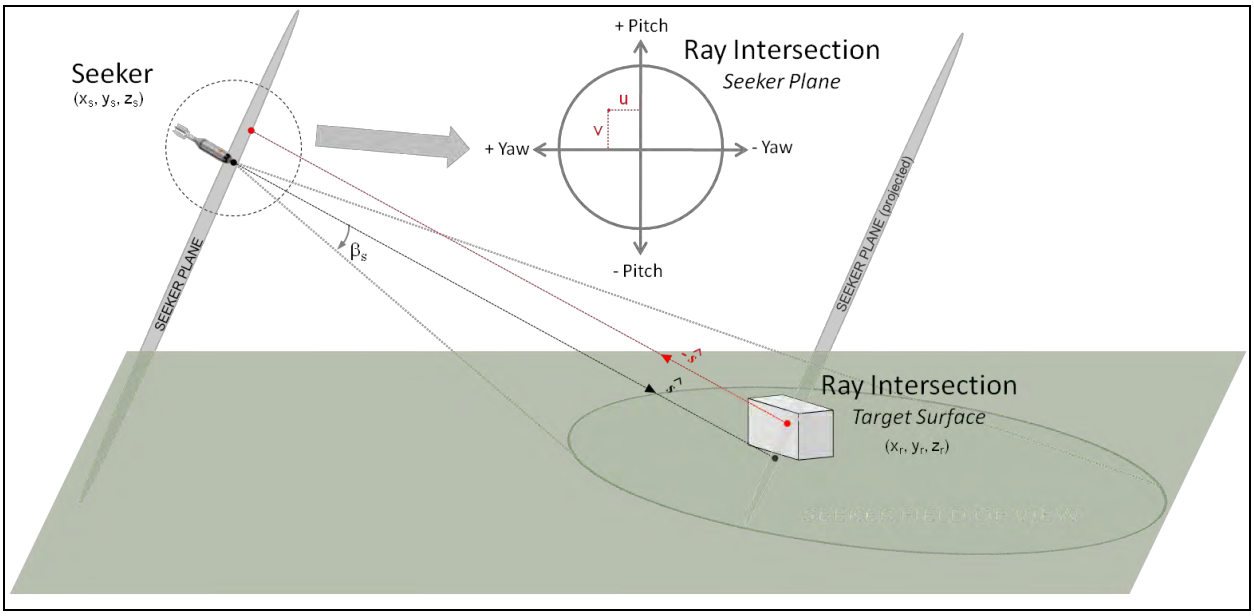


Figure 13. Ray intersection with seeker plane.

3.3.2 Power Received by Seeker

After recording what quadrant the ray projects into, the model determines the ray's individual power contribution to the quadrant. To determine the amount of power received by the seeker for each ray, the model uses Lambert's Cosine Law. The law states that the radiant intensity I (power per steradian) received from a perfectly Lambertian surface is proportional to the cosine of the angle, α, between the observer's line of sight and the surface normal (figure 1, equation 25).

In reference 6, McCartney describes how to calculate the radiant flux ϕ (power) across a hemisphere by integrating the radiant intensity, I (figure 14). In the hemisphere, α represents the 90° complement to elevation, and ω represents azimuth.* From the conservation of energy, we know that the total power in the system passing through the hemisphere is equal to the total power reflecting off of the target (neglecting attenuation losses, which will be factored in later). Therefore, if we integrate over the hemisphere of intensity $I_{peak} \cdot \cos\alpha$, we get the total power passing through the hemisphere, ϕ (equations 26–27). Because we know ϕ as the reflected power off of the target, we can solve for the hemisphere's peak radiant intensity, I_{peak} (equation 28). Finally, using Lambert's Cosine Law, we determine the radiant intensity at the seeker's position on the hemisphere (equation 29).

*Note: α and ω are different from the seeker azimuth (ϕ) and elevation (θ). α and ω characterize the hemisphere extending from the designated surface and the seeker's *position* on this hemisphere. ϕ and θ describe the *orientation* of the seeker in the global coordinate system.

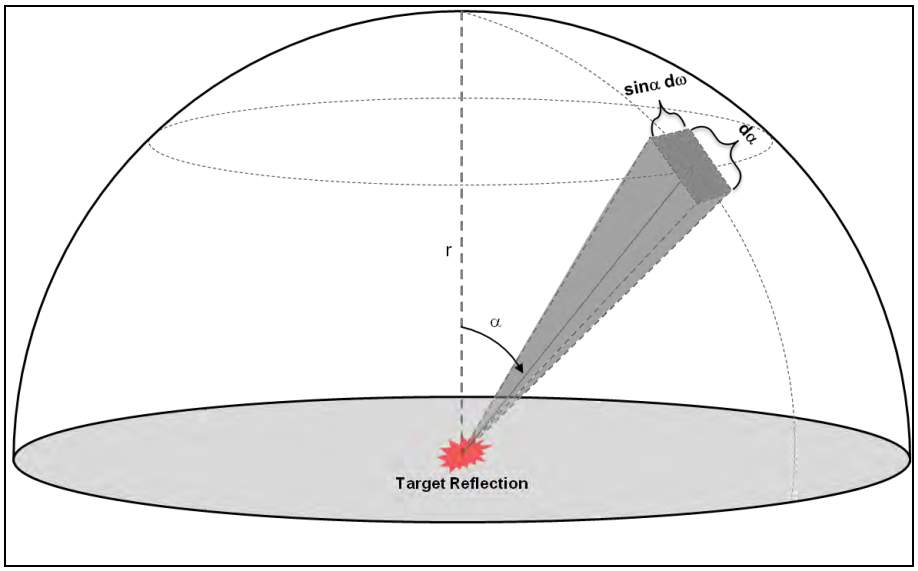


Figure 14. Integration of radiant intensity over a hemisphere.

$$\phi = \int_0^{\pi/2} \int_0^{2\pi} I_{peak} cos\alpha (sin\alpha) d\omega d\alpha. \tag{26}$$

Substituting a double angle:

$$\phi = I_{peak} \int_0^{\pi/2} \int_0^{2\pi} \frac{1}{2} sin(2\alpha) d\omega d\alpha$$

$$\phi = \pi \cdot I_{peak} \int_0^{\pi/2} sin(2\alpha) d\alpha$$

$$\phi = \pi \cdot I_{peak}. \tag{27}$$

Therefore, solving for the peak radiant intensity in the hemisphere:

$$I_{peak} = \frac{\phi_{target,reflected}}{\pi}. \tag{28}$$

Using Lambert's Cosine Law, we determine the radiant intensity at the seeker's location on the hemisphere:

$$I_{seeker} = I_{peak} \cdot \cos \alpha.$$

$$I_{seeker} = \frac{\phi_{target,reflected}}{\pi} \cdot \cos \alpha. \tag{29}$$

To determine the power at the seeker (before factoring in other losses), we multiply I_{seeker} by the solid angle subtended by the seeker on the hemisphere. In the model, the hemisphere extends from the laser target intersection point normal to the surface of intersection. The radius of the hemisphere, r, is the range from the laser-target intersection point to the geometric center of the seeker. To approximate the number of steradians the seeker subtends on the hemisphere

(assuming $r_{seeker} << r$), the seeker area, S_{seeker}, is divided by r^2 and multiplied by the cosine of the seeker correction angle, μ (figure 1, equation 16):

$$\Omega_{seeker} \cong \frac{S_{seeker}}{r^2} \cdot \cos\mu \tag{30}$$

Thus, including attenuation from target hit to seeker (L_{Atten}), and seeker efficiency ($s_{efficiency}$), the laser power scattered from the target into the seeker per ray is

$$\phi_{seeker} = I_{seeker} \cdot \Omega_{seeker} \cdot L_{Atten} \cdot s_{efficiency}$$

$$\phi_{seeker} = \frac{\phi_{target,reflected} \cdot \cos\alpha \cdot S_{seeker} \cdot \cos\mu \cdot L_{Atten} \cdot s_{efficiency}}{\pi r^2} \tag{31}$$

Equation 31 is calculated for each ray. The model sums the individual ray contributions for each quadrant to find the total power received in all four seeker quadrants. The next section discusses how the model uses these data to produce guidance signals.

4. Seeker Guidance Model

In the last section, the model calculated the beam's transmission path and power loss, resulting in a power distribution at the seeker. In this section, we discuss how the model interprets these data to produce projectile guidance.

4.1 Target Encounter and Detect

There exists a progressive ladder in the target knowledge of a terminal seeker (figure 15).

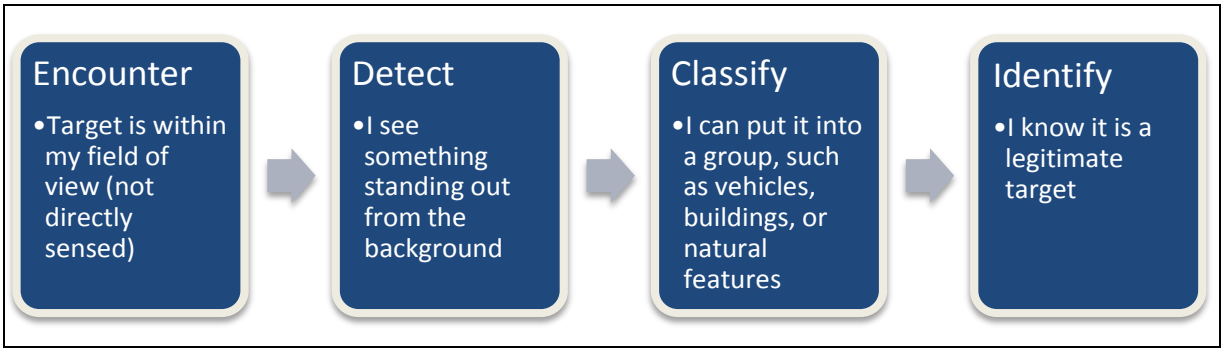


Figure 15. Target knowledge ladder.

For the SAL seeker, the knowledge ladder is modeled in the following way:

- *Encounter* is determined in the model by calculating the angle the seeker needs to rotate through to point at the target's center. If the angle is less than or equal to the FOV, it returns true. *Encounter* is internal to this model, as it is a geometric calculation and is not

sensed by a real projectile. It is a useful calculation, however, to determine the point in the ballistic trajectory at which the seeker is pointing close enough to the target to encounter, which is the first step toward detection.

- *Detection* occurs in the model when the laser power the seeker receives is above the S/N threshold necessary to distinguish the signal. The noise is determined using a background noise multiplier, which assumes that the seeker noise increases linearly with seeker area and solid angle subtended by the FOV. This also assumes that the seeker internal noise is insignificant compared to the external background noise, as internal noise does not scale with seeker area or solid angle. The model's default multiplier value is based on a commercial seeker with minimum detectable signal irradiance (power per unit seeker area) of 35nW/cm^2, FOV half angle of 4.5˚, and an assumed S/N ratio of 7 (7).
- *Classify and Identify* are combined in the final step, and occur when the SAL seeker examines the signal's pulse width and pulse frequency to weed out false signals. Because the model does not contain a progressive time element, these effects are not modeled, but could easily be added upon integration into a larger guided trajectory program.

4.2 Guidance Updates

After the seeker detects, it needs to process the information into guidance signals.* It does this by comparing the signals in each of the four sections of the detector. Figure 16 shows the four-quadrant detector used in the model.

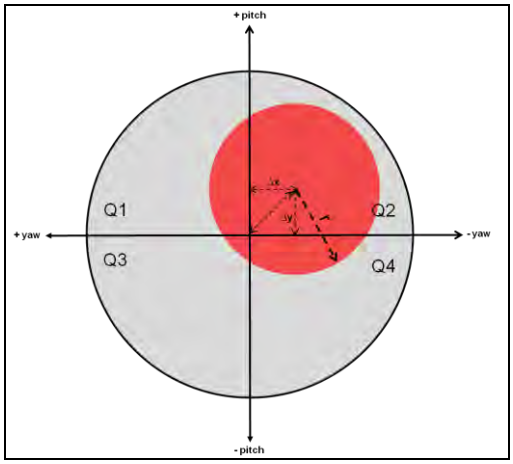


Figure 16. Four-quadrant laser detector.

*While the model always computes guidance signals as a convenience to the user, they should not be considered reliable unless also accompanied by a positive detect calculation.

If P_1 is the power received in quadrant 1, and likewise for the other quadrants, Hubbard describes how the spots' location can be approximated in the following form (*8*):

$$\frac{\Delta y}{r} \cong \frac{(P_1+P_2)-(P_3+P_4)}{P_{total}} \rightarrow Pitch\ Guidance, \tag{32}$$

$$\frac{\Delta x}{r} \cong \frac{(P_1+P_3)-(P_2+P_4)}{P_{total}} \rightarrow Yaw\ Guidance, \tag{33}$$

$$where\ P_{total} = P_1 + P_2 + P_3 + P_4. \tag{34}$$

The pitch and yaw guidance range in value from –1 to +1, which mean full maneuvers in the negative and positive directions, respectively. In calculating guidance, the model assumes a non-spun projectile, although spin could be accommodated in a fairly simple manner upon integration into a larger, time-dependent model.

The model also returns a polar representation of where the power is received on the sensor:

- Angle (θ): 0 to 2π rad, 0 = –Yaw direction, angle increases in the counter-clockwise (CCW) direction.
- Magnitude: 0 to 1.

The angle determines how to combine pitch and yaw commands and the magnitude determines how much to maneuver in the prescribed direction. The polar output is currently a repackaged version of the preceding pitch and yaw guidance, but could be a better representation to use in the future for imaging sensors, in order to track multiple data points within a single FOV.

5. C++ Implementation

5.1 Input Variables

The input variables (table 1) usually change throughout an analysis, and are defined in the main.cpp file.

Table 1. Input variables.

Locations	Type	Units	Bounds	Description
d_geo	struct Point3D *Utilities* class double	m	z > = 0	**Designator Location:** *Global Coordinates, Point Mass*
t_geo	Point3D	m	z > = 0	**Target Location:** *Global Coordinates, Geometric Center* Due to flat terrain, the ground plane is located at z = 0. If the target is ground-based, its z component must be given accordingly (z = 0 + 1/2 target height).
s_geo	Point3D	m	z > = 0	**Seeker Location** *Global Coordinates, Geometric Center*
Orientations	Type	Units	Bounds	Description
s_orientation	struct Point2D *Utilities* class double	rad	Az: -π to π El: -π/2 to π/2	**Seeker Orientation***: Global Coordinates* **.X (Azimuth, ϕ)** – the angle between the x axis and the projection of $\hat{s}$ onto the x-y plane. Azimuth is positive in the counter-clockwise direction as seen from above. **.Y (Elevation, θ)** – the angle between the x-y plane and $\hat{s}$. Elevation is positive when $\hat{s}$ pointins above the x-y plane.
t_rotation	double	rad	0–2π	**Target Rotation***: Global Coordinates* **τ =** the angle between the +y axis and the projection of the target heading vector, $\hat{t}$, onto the x-y plane measured clockwise as seen from above. $\hat{t}$ points from the target geometric center towards the center of the front face of the target.
Statistics	Type	Units	Bounds	Description
seed	int	—	—	Seed for random number generator

5.2 Input Parameters

The parameters (table 2) tune the analysis, but generally do not vary during a simulation. They can be set in the main.cpp file, or through an input file (see appendices A and B, main.cpp #2). If no change in a parameter is detected, the program uses the default parameter value, which is automatically set using the internal function SetDefaults() when the class is first instantiated.

Table 2. Input parameters.

Weather	Type	Units	Bounds	Default	Description
sal_weather	struct weather int	—	Seas: 0-2 Lat: 0-1 Vis: 0-1	Seas: 0 Lat: 1 Vis: 0	.season 0 = summer (March 22 – September 21) 1 = winter (September 22 – March 21) .latitude 0 = tropics (0° to 23.5° ±) 1 = mid-latitudes (23.5° to 50° ±) 2 = sub-arctic (50° to 70° ±) .visibility 0 = clear (23 km) 1 = hazy (5 km)
Seeker	Type	Units	Bounds	Default	Description
s_fov	double	rad	0 - π/2	0.079 (4.5˚)	Seeker field of view (half angle of cone)
s_aperture _diameter	double	m	> 0	0.060	Seeker aperture diameter
s_background	double	$Wm^{-2}\Omega^{-1}$	> = 0	0.010	Seeker background noise multiplier
s_signalnoise _threshold	double	—	> 0	7.0	Ratio of SAL signal to background noise to be able to detect
s_efficiency	double	—	0–1	0.95	Seeker loss coefficient
Designator	Type	Units	Bounds	Default	Description
d_lasertype	int	—	0–1	0	0 = 1.06 µm (Nd:Glass) 1 = 1.536 µm (Er:Glass)
d_raycount	int	—	> = 1	10000	No. of rays to divide laser pulse into
d_divergence	double	rad	> = 0	3E-4	Divergence of laser beam (half angle)
d_pulse_energy	double	J	> 0	80E-3	Starting designator energy/pulse
d_pulse_frequency	double	Hz	> 0	10	No. of pulses/second
d_pulse_duration	double	s	> 0	15E-9	Pulse length
d_efficiency	double	-	0 – 1	0.95	Designator loss coefficient
d_h_error	double	rad	> = 0	1E-4	Designator pointing error (horizontal, SD)
d_v_error	double	rad	> = 0	1E-4	Designator pointing error (vertical, SD)
Target	Type	Units	Bounds	Default	Description
t_size	struct Point3D *Utilities* class double	m	> 0	6.4,2.3,2.3	Target Dimensions .X = length .Y = width .Z = height
t_reflect	array[int]	—	0–1	0.4 for all	Target Reflectivity [0] = left side, [1] = front, [2] = right side [3] = back, [4] = top, [5] = ground plane

5.3 Output

The user only needs to call one function, S_SALSEEKER, which performs all necessary calculations and outputs all results to a class-defined struct, "saloutputs" (see table 3).

Table 3. The S_SALSEEKER output function.

	Type	Units	Description
S_SALSEEKER (d_geo, t_geo, s_geo, s_orientation, t_rotation, seed)	struct saloutputs	—	*Main calculation function* **OUTPUT** **.encounter (bool):** target is within seeker field of view **.detect (bool):** S/N above threshold **.sal_signals (struct):** magnitude of peak laser power sensed in each quadrant (W) **.q1** = Positive Pitch, Positive Yaw **.q2** = Positive Pitch, Negative Yaw **.q3** = Negative Pitch, Positive Yaw **.q4** = Negative Pitch, Negative Yaw **.actuator_signals (struct):** lifting and turning guidance based on *sal_signals* quadrant values **.Pitch, .Yaw (-1 to 1)** -1 is full maneuver in negative direction (-yaw, -pitch) +1 is full maneuver in the positive direction. Partial maneuvers for numbers in between -1 and +1. **.Theta, .Mag (polar representation of signal in FOV)** .Theta = Polar direction of signal center (0-2π rad) 0 rad = - yaw direction, rotate CCW) .Mag – Polar magnitude of direction vector (0-1)

6. Validation

All validation tests were run using the default parameters unless otherwise noted (default parameters listed in section 5.2).

6.1 Power Loss

The NVLaserD model was used to validate the power drop of the laser across a given distance (*9*) (see table 4). The beam divergence (δ) was varied from 0 to 2.4 milliradians (mrad) and for each divergence, the seeker angle from the target surface was tested at 0° and 45°. When δ equaled 0, the results agreed within 1%. As δ increased, the results originally differed. This disagreement is most likely explained by a difference in the way δ is defined between models. Recall equation 13 ($\sigma_\delta \cong \delta/2$), which assumed a δ corresponding to a beam radius of 2 σ from the peak intensity. If δ was instead assumed to correspond to a 1-σ beam radius, the resulting beam spread would be twice as great as the 2-σ beam spread (figure 17).

Table 4. Energy drop comparison between sSalSeeker and NVLaserD.

TEST INPUTS	TEST 1		TEST 2		TEST 3		TEST 4	
Designator Pulsed Energy (mJ)	80	80	80	80	80	80	80	80
Designator Pulse Width (ns)	15	15	15	15	15	15	15	15
Designator Pulsed Power (Peak - MW)	5.3	5.3	5.3	5.3	5.3	5.3	5.3	5.3
Designator Error (mrad)	0	0	0	0	0	0	0	0
Target Rotation (°)	0	0	0	0	0	0	0	0
Range (Designator to Target, km)	2	2	2	2	2	2	2	2
Range (Seeker to Target, km)	5	5	5	5	5	5	5	5
k (attenuation coefficient)	0.0581	0.0581	0.0581	0.0581	0.0581	0.0581	0.0581	0.0581
δ (divergence - mrad) – s_SalSeeker	0	0	0.6	0.6	1.2	1.2	2.4	2.4
δ (divergence - mrad) – NVLaserD	0	0	0.3	0.3	0.6	0.6	1.2	1.2
α (seeker to target normal angle - deg)	0	45	0	45	0	45	0	45
SEEKER SIGNAL (X 10^{-5} W)								
sSalSeeker	**4.59**	**3.25**	**4.34**	**3.07**	**2.98**	**2.11**	**1.35**	**0.96**
NVLaserD	**4.62**	**3.26**	**4.36**	**3.08**	**3.03**	**2.14**	**1.39**	**0.98**
Difference (%)	**–0.7**	**–0.3**	**–0.5**	**–0.3**	**–1.7**	**–1.4**	**–2.9**	**–2.1**

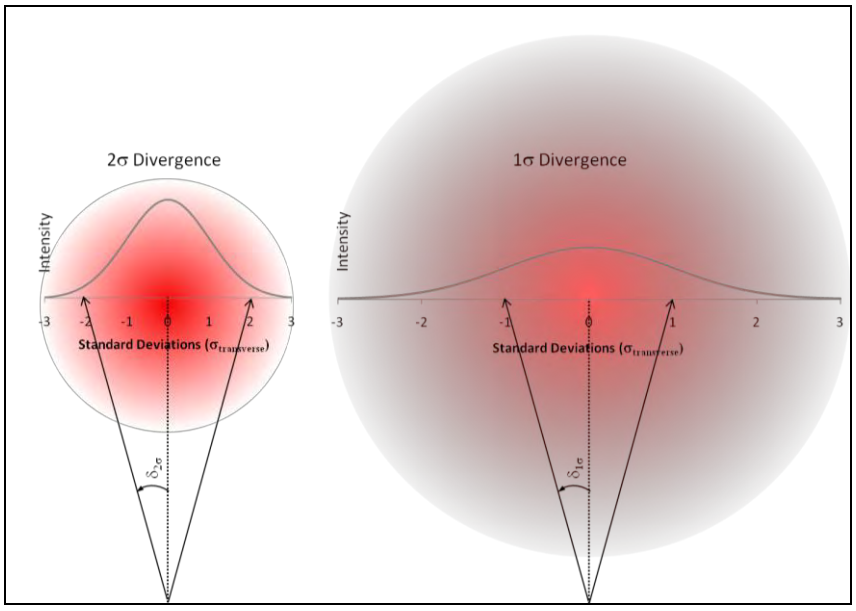


Figure 17. Comparison of 2-σ and 1-σ divergence.

This theory is supported by the NVLaserD beam dimensions at the target, whose standard deviations were twice those predicted by sSalSeeker. Therefore, to compare calculations with the NVLaserD model, the sSalSeeker δ values were made double those of NVLaserD. After making this assumption, the models agreed within 3% for all comparisons.

6.2 Geometry

6.2.1 Projectile Fly-In

Initial Conditions

- Target Location (t_{geo}) = (0,0,1.15)
- Designator Location (d_{geo}) = (–2000,0,1.15)
- Seeker Location (s_{geo}) = (–25000,0,25000)
- Seeker Azimuth (s_orientation.X) = 0°
- Seeker Elevation (s_orientation.Y) = –45°
- Target Rotation (t_rotation) = 0°

Variables

- Seeker Location (s_{geo}) = (x,0,z) – varied so that seeker is on a 45° descent towards target
- Visibility (sal_weather.visibility) = clear, hazy

Figure 18 shows the results of a virtual fly-in that tested how much the laser signal increased as the range to target decreased. The projectile began approximately 35 km away from the target (25 km horizontally and 25 km vertically), and closed in on the target at a 45° descent. Designator, projectile, and target were all aligned along the x-axis (no side-to-side movement). The green line in figure 18 represents the S/N threshold above which the seeker detects. For the fly-in test, the seeker detected at a 21.5-km horizontal range for clear conditions and at 14.25-km horizontal range for hazy conditions.

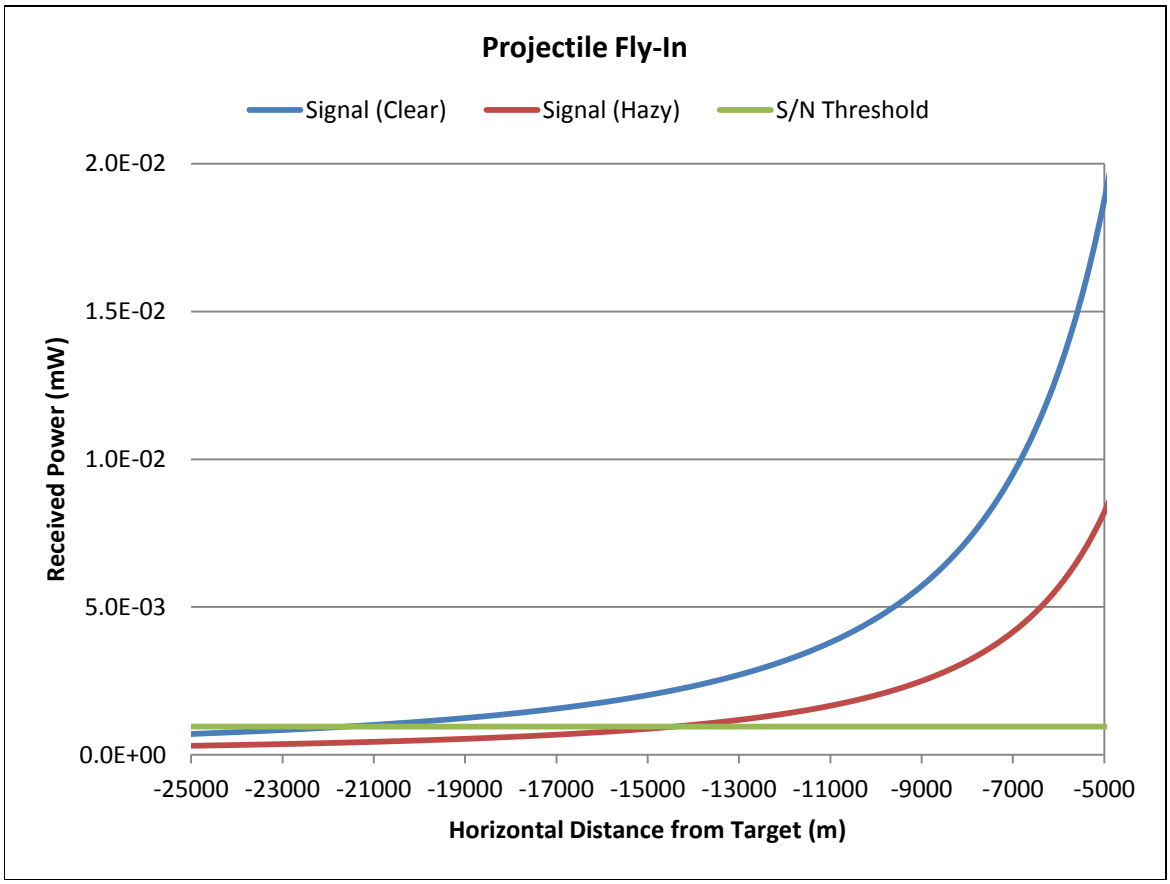


Figure 18. Projectile fly-in test.

6.2.2 Off-Angle Test

Initial Conditions

- Target Location (t_{geo}) = (0,0,1.15)
- Designator Location (d_{geo}) = (–2000,0,1.15)
- Seeker Location (s_{geo}) = (–2000,0,1.15)
- Seeker Azimuth (s_orientation.X) = 0°
- Seeker Elevation (s_orientation.Y) = 0°
- Target Rotation (t_rotation) = 0°

Variables

- Seeker Location (s_{geo}) = varied so that x and y are on circle with radius of 2000 m
- Seeker Azimuth (s_orientation.X) = varied so that seeker always points at the geometric center of the target

The seeker was positioned at a series of points on a circle around the parallelepiped target, always pointing at the target's geometric center. The targets' location was fixed and its rotation was 0. The designator remained fixed, and aimed along the +x axis to designate the left side of the target. As the seeker's y deviated from 0, the seeker's off-angle from the target surface normal (α, see figure 1) increased from 0° to 90° in 3-degree increments, and the received power decreased as the cosine of that angle (figure 19). At 90°, the seeker faced parallel to the designated surface of the target, at the boundary of the hemisphere containing the reflected laser power. Beyond 90°, the seeker moved outside of this hemisphere, and the received power dropped to 0. Points along the graph's horizontal axis are seeker positions where α was beyond 90°.

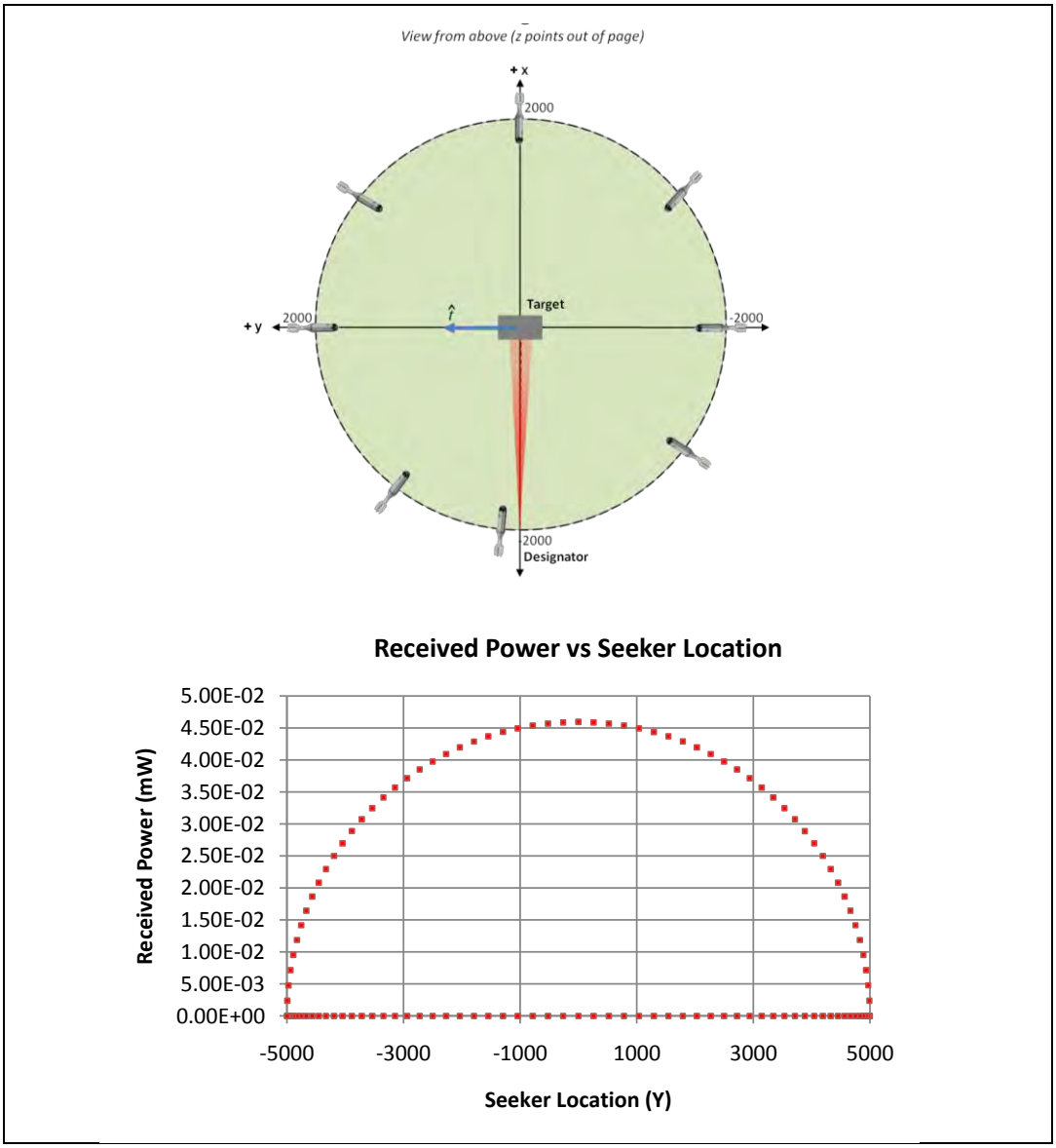


Figure 19. Off-angle test.

6.2.3 Target Rotation Test

Initial Conditions

- Target Location (t_{geo}) = (0,0,1.15)
- Designator Location (d_{geo}) = (–2000,0,1.15)
- Seeker Location (s_{geo}) = (–5000,0,1.15)
- Seeker Azimuth (s_orientation.X) = 0°
- Seeker Elevation (s_orientation.Y) = 0°
- Target Rotation (t_rotation) = 0°

Variables

- Target Rotation (t_rotation) = 0 to 360°
- Divergence (d_divergence) = 0.3, 0.6 mrad

While the designator and seeker stayed at a fixed distance from the target, the target rotation was varied from 0° to 360°. Figure 20 shows how the power varied for different target rotation angles, with the rectangular target (6.4 × 2.3 m) displaying wider variation than the square target (2.3 × 2.3 m). For δ = 0.3 mrad, the rectangular target underperformed the square target at target rotation angles such as 60°. This occurred when the rectangle's long surface was the main side receiving power, but at a very oblique off-angle (α) from the seeker. This observation poses an interesting question for the designator aim point, whether it is best to point at the target center, or at the side most normal to the beam. As beam divergence increased, the larger rectangular target improved its performance relative to the square target, as the beam started to spill off of the sides of the smaller square target (note the graph for δ = 0.6 mrad, where the rectangular target outperformed the square target at angles of 0° and 180°).

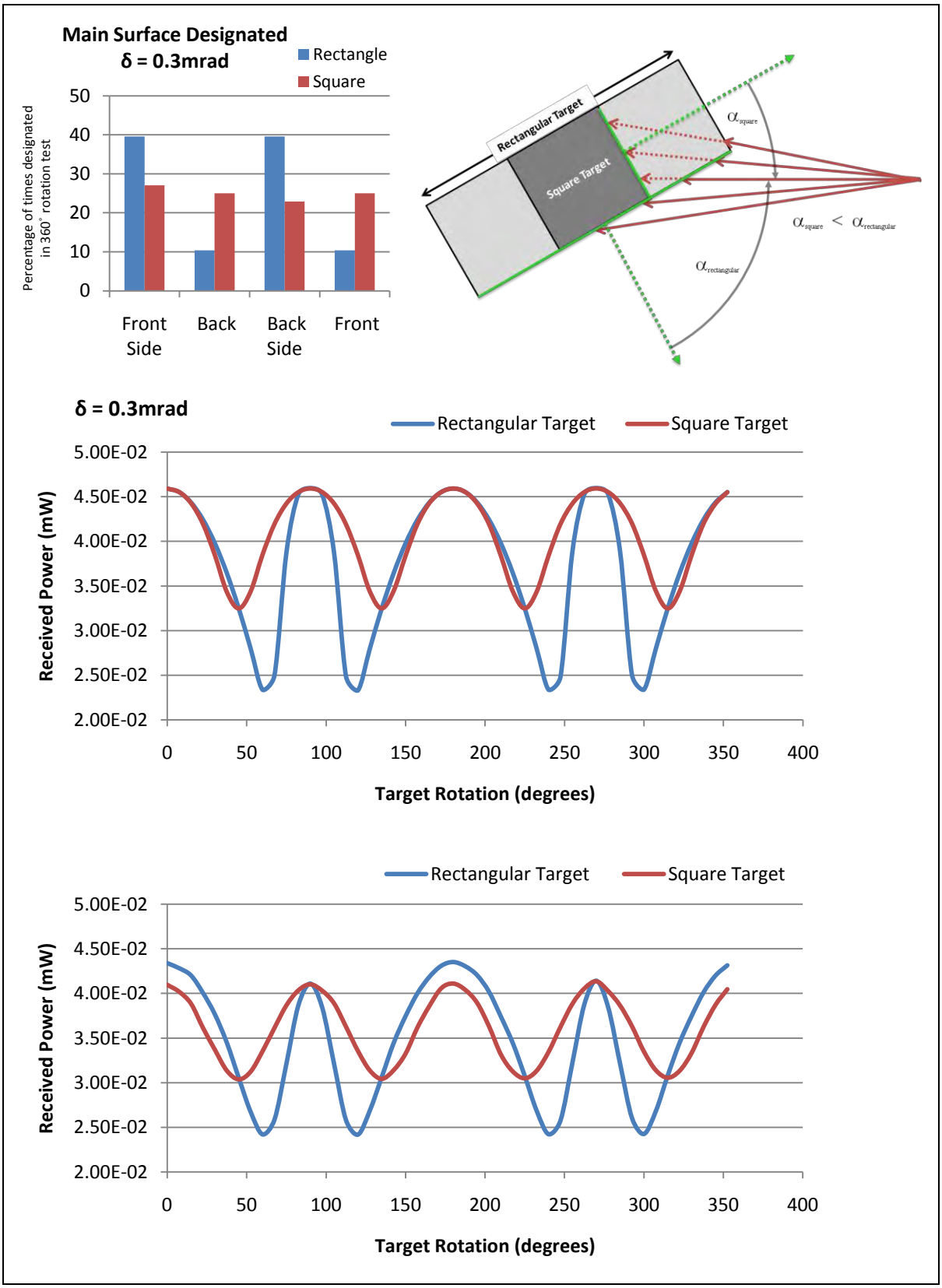


Figure 20. Target rotation test.

7. Path Forward

In summary, the model calculates the power distribution across a SAL seeker's IR detector given relevant geometry, laser characteristics, and atmospheric information. By analyzing this power distribution, the model returns flight guidance information. The model's C++ class implementation, *sSalSeeker*, can be run stand-alone, and is also easily embeddable into larger smart-weapon models.

Recently, researchers have investigated the ability of reduced-state guidance algorithms to successfully guide a munition to the target (*10*). Upon integration into a larger smart-weapon model, this model could provide insight into another simplified guidance scenario: SAL-only guidance, without input from a global-positioning system (GPS) or inertial measurement unit (IMU). This requires the ability to shoot ballistically into a "guidance basket," after which the SAL seeker takes over (figure 21). The dimensions of the basket depend largely on the seeker's FOV, the rate at which the laser beam attenuates, and the maneuver authority of the projectile.

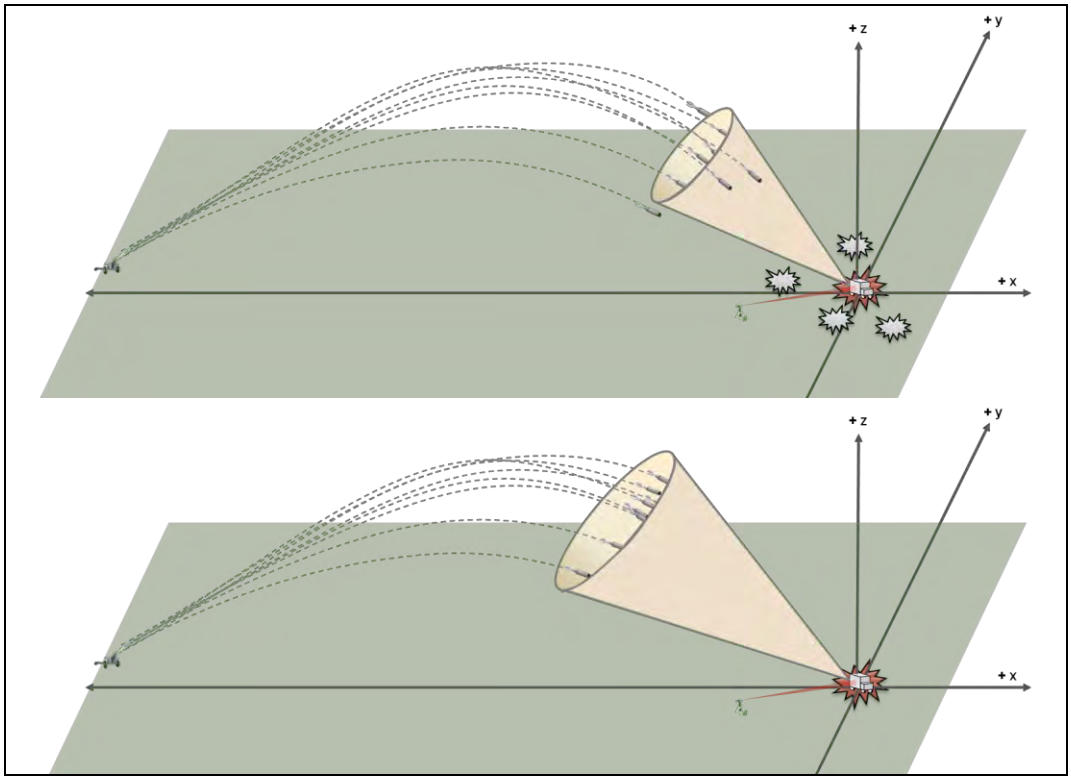


Figure 21. Extending the guidance basket.

The following are suggestions for further development of the model:

- **Target geometry**
  Current – Rectangular parallelepiped
  Future – Detailed target geometries including windows and tires would permit study of whether there is an ideal location to designate for different targets.
  Difficulty – Low

- **Terrain**
  Current – Flat terrain, which makes for an unrealistically-favorable situation.
  Future – Variable-height terrain, with vegetation and man-made features
  Difficulty – Medium

- **Target reflection**
  Current – Assumes perfect Lambertian surface reflection
  Future – Combination of Lambertian and specular reflection, depending on the surface. Reflection would then depend on the angle of incidence from the designator. For example, as target geometries become more developed, windows may incorporate more specular reflection than matte-finished doors.
  Difficulty – Medium

- **Background noise**
  Current – Input parameter
  Future – Calculation of the background noise from the sun and earth entering the seeker
  Difficulty – Medium

8. References

1. Yager, R. J. *A Plant Model For Smart Projectiles*; ARL-TR-5520; U.S. Army Research Laboratory: Aberdeen Proving Ground, MD, 2011.

2. Weichel, H. *Laser Beam Propagation in the Atmosphere*; SPIE: Bellingham, **1990**.

3. McClatchey, R.A. *Optical Properties of the Atmosphere (Revised)*; AFCRL-71-0279; Air Force Cambridge Research Laboratories: L.G. Hanscom Field, Bedford, MA, 1971.

4. Line-Plane Intersection. http://en.wikipedia.org/wiki/Line-plane_intersection (accessed March 20, 2011).

5. Jones, R. F. *Survey of Laser Reflectivity Measurements at 1.06 Microns*; RE-72-17; U.S. Army Missile Command: Redstone Arsenal, AL, 1972.

6. McCartney, E. *Optics of the Atmosphere: Scattering by Molecules and Particles*; John Wiley and Sons: New York, 1976.

7. Selex Galileo. Laser Spot Tracker 2nd Generation Specification Sheet. http://www.selex-sas.com/EN/Common/files/SELEX_Galileo/Products/LST_gen2_dsh82.pdf (accessed July 13, 2011).

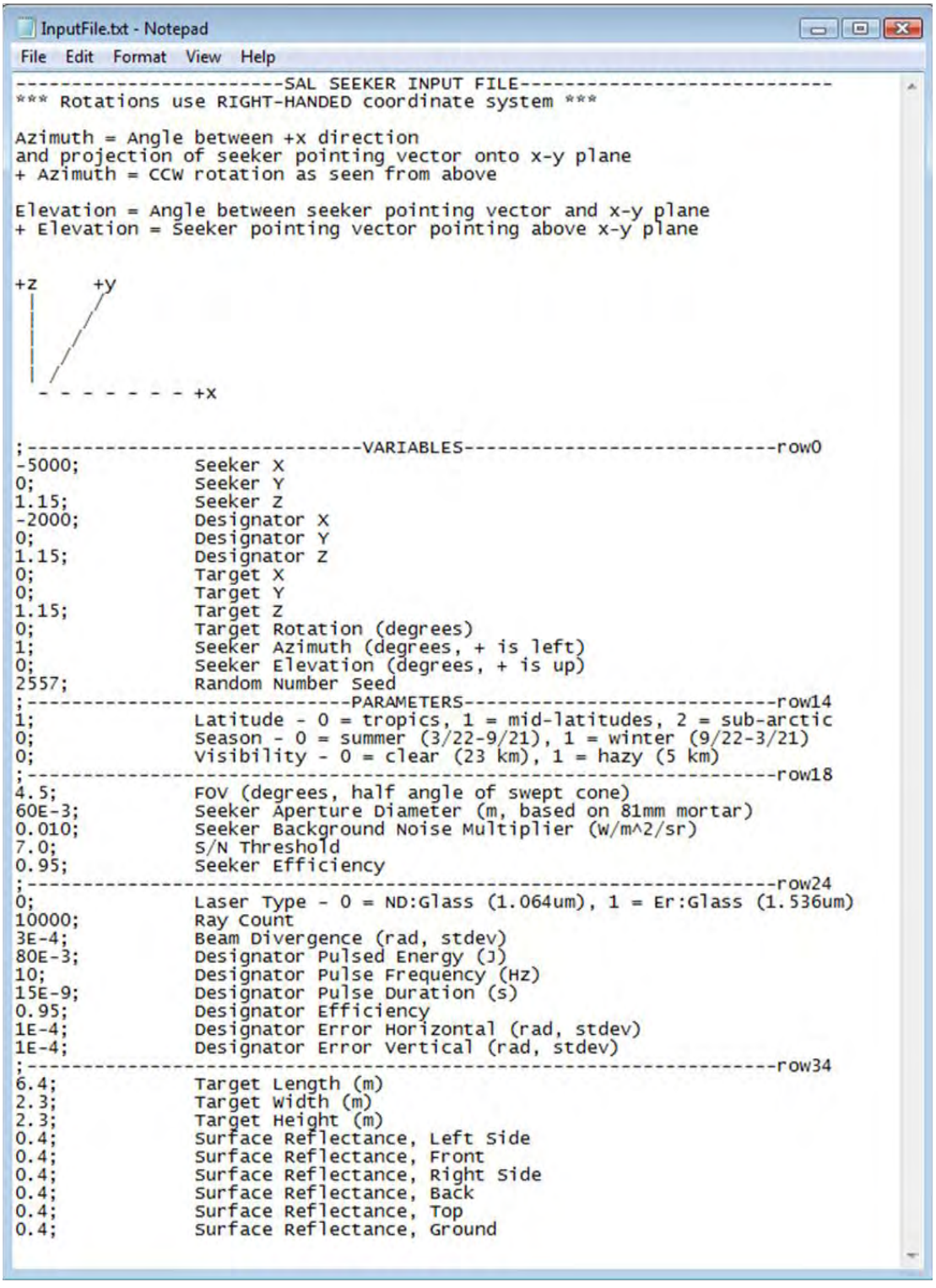
8. Hubbard, K. A. *Characterization of Semi-Active Laser (SAL) Seekers for Affordable Precision Guidance of Gun-Launched Munitions*; ARL-TR-5233; U.S. Army Research Laboratory: Aberdeen Proving Ground, MD, 2010.

9. *NVLaserD, Night Vision Laser Designator Model*, Version 1.0; Night Vision and Electronics Sensors Directorate: Fort Belvoir, VA, 2006.

10. Fresconi, F. *Guidance and Control of a Fin-Stabilized Projectile Based on Flight Dynamics With Reduced Sensor and Actuator Requirements*; ARL-TR-5458; U.S. Army Research Laboratory: Aberdeen Proving Ground, MD, 2011.

Appendix A. Sample Input and Output

This appendix is in its original form, without editorial change.



InputFile.txt - Notepad

File Edit Format View Help

```
----------------------SAL SEEKER INPUT FILE---------------------------
*** Rotations use RIGHT-HANDED coordinate system ***

Azimuth = Angle between +x direction
and projection of seeker pointing vector onto x-y plane
+ Azimuth = CCW rotation as seen from above

Elevation = Angle between seeker pointing vector and x-y plane
+ Elevation = Seeker pointing vector pointing above x-y plane


+z     +y
 |    /
 |   /
 |  /
 | /
  - - - - - - - +x


;-----------------------------VARIABLES-----------------------------row0
-5000;          Seeker X
0;              Seeker Y
1.15;           Seeker Z
-2000;          Designator X
0;              Designator Y
1.15;           Designator Z
0;              Target X
0;              Target Y
1.15;           Target Z
0;              Target Rotation (degrees)
1;              Seeker Azimuth (degrees, + is left)
0;              Seeker Elevation (degrees, + is up)
2557;           Random Number Seed
;----------------------------PARAMETERS----------------------------row14
1;              Latitude - 0 = tropics, 1 = mid-latitudes, 2 = sub-arctic
0;              Season - 0 = summer (3/22-9/21), 1 = winter (9/22-3/21)
0;              Visibility - 0 = clear (23 km), 1 = hazy (5 km)
;-------------------------------------------------------------------row18
4.5;            FOV (degrees, half angle of swept cone)
60E-3;          Seeker Aperture Diameter (m, based on 81mm mortar)
0.010;          Seeker Background Noise Multiplier (W/m^2/sr)
7.0;            S/N Threshold
0.95;           Seeker Efficiency
;-------------------------------------------------------------------row24
0;              Laser Type - 0 = ND:Glass (1.064um), 1 = Er:Glass (1.536um)
10000;          Ray Count
3E-4;           Beam Divergence (rad, stdev)
80E-3;          Designator Pulsed Energy (J)
10;             Designator Pulse Frequency (Hz)
15E-9;          Designator Pulse Duration (s)
0.95;           Designator Efficiency
1E-4;           Designator Error Horizontal (rad, stdev)
1E-4;           Designator Error Vertical (rad, stdev)
;-------------------------------------------------------------------row34
6.4;            Target Length (m)
2.3;            Target Width (m)
2.3;            Target Height (m)
0.4;            Surface Reflectance, Left Side
0.4;            Surface Reflectance, Front
0.4;            Surface Reflectance, Right Side
0.4;            Surface Reflectance, Back
0.4;            Surface Reflectance, Top
0.4;            Surface Reflectance, Ground
```

Figure A-1. Sample input file.

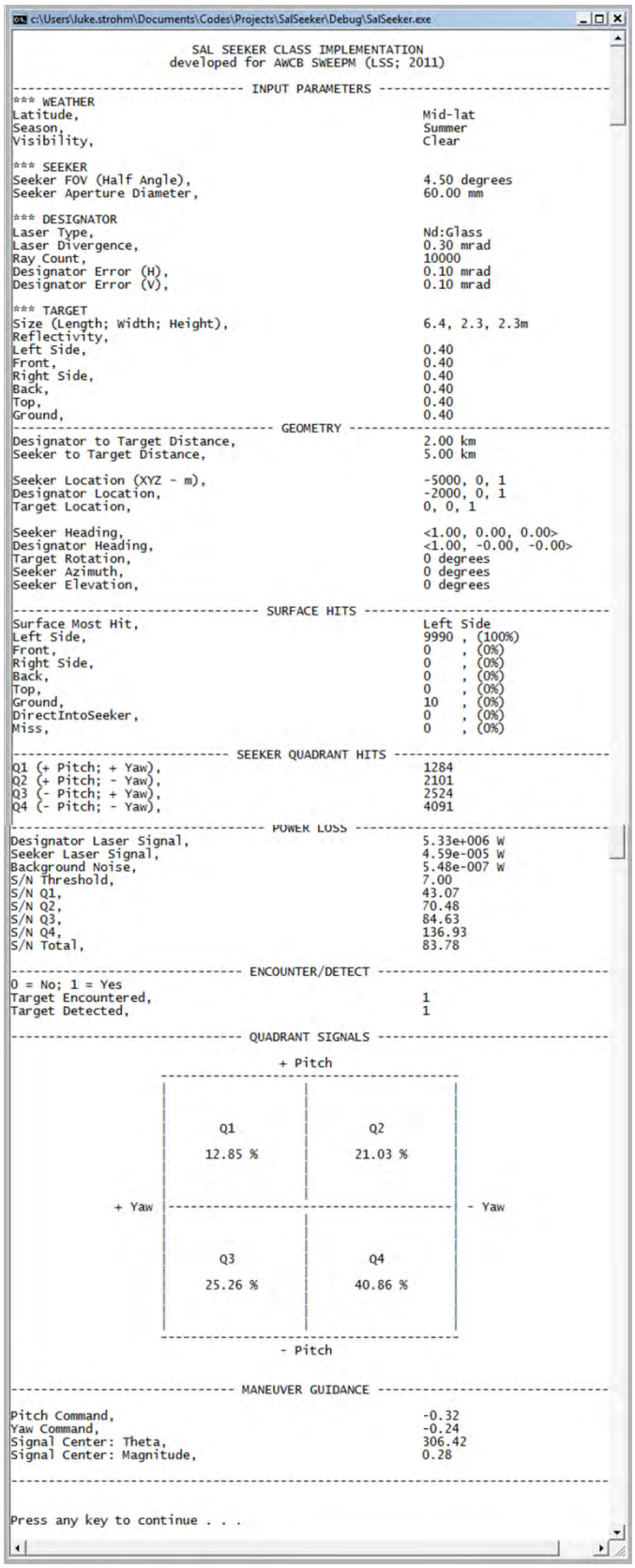


Figure A-2. Sample output file.

INTENTIONALLY LEFT BLANK.

Appendix B. sSalSeeker Code

This appendix is in its original form, without editorial change.

s_sal_seeker_class.h

```
/*
SAL Seeker Class .h
Luke Strohm, Army Research Laboratory
3-14-11
*/

/*****************************************************************************/
#ifndef S_SAL_SEEKER_CLASS_H_
#define S_SAL_SEEKER_CLASS_H_
/*****************************************************************************/
#include <iostream>
#include <iomanip>
#include <io.h>
#include <fstream>
#include <string>
#include <cstdlib>
#include <windows.h>
#include <cmath>
/*****************************************************************************/
#include "utilities.h" //.....................................MKA Utility Library
#include "statistics.h" //...................................MKA Statistics Library
#include "y_format_namespace.h" //.........................RJY Formatting Library
using namespace std;
using namespace yFormat;
/*****************************************************************************/
//  Define Structs
struct weather{
  int season; //.............................................0 = summer (3/22-9/21)
              //.............................................1 = winter (9/22-3/21)
  int latitude; //.....................................0 = tropics (<23.5 deg +-)
                //.............................1 = mid-latitudes (23.5-50 deg +-)
                //.................................2 = sub-arctic (50-70 deg +-)
  int visibility; //.................................0 = clear (23 km visibility)
                  //.....................................1 = hazy (5 km visibility)
};
struct rayhit{
  Point3D hit; //...................X,Y,Z Coords of where ray intersects surface
  int surf; //.....................................................Surface point hits
  double power; //.....................................................Power of ray
  bool obscured; //.Check to see if ground hit is obscured from seeker by target
};
struct quadsignal{ //......Contains info on the four quads of quad-band detector
  double q1; //.....................................Positive Pitch, Positive Yaw
  double q2; //.....................................Positive Pitch, Negative Yaw
  double q3; //.....................................Negative Pitch, Positive Yaw
  double q4; //.....................................Negative Pitch, Negative Yaw
};
struct guidancesignal{ //..........Actuator guidance (-1/+1 is maximum maneuver)
  double pitch; //.......................Pitching (Lifting+) Multiplier (-1 to 1)
  double yaw; //.........................Yawing (Left Turn+) Multiplier (-1 to 1)
  double theta; //.............................Polar Direction of Signal Center
                //.............................0-2pi rad, 0 = -yaw dir, rotate CCW
  double mag; //.......................Polar Magnitude of Direction Vector (0-1)
};
struct saloutputs{
  bool encounter;
  bool detect;
  quadsignal sal_signals;
  guidancesignal actuator_signals;
};
/*****************************************************************************/
class sSalSeeker{
```

```
public://************************---PUBLIC---************************************

  //****************************OUTSIDE CLASSES*********************************
  Utilities UTILITY; //.........................Instantiate Utility Class (MKA)
  Statistics STAT; //.........................Instantiate Statistics Class (MKA)
  //*******************************OUTPUT*****************************************
  //O_FUNCTIONS-------------------------------------------------------------------
  sSalSeeker();
  saloutputs S_SALSEEKER(Point3D d_geo, Point3D t_geo, //..Runs all calculations
                 Point3D s_geo, Point2D s_orientation,//.and outputs to struct
                 double t_rotation, int seed);
  saloutputs programoutput;
  void PrintOutput(string filename = "", string format = ".txt"); // SummaryFile
  //O_DISPLAY---------------------------------------------------------------------
  bool printtoscreen;
  bool printtofile;
  int surface[8]; //.0 = Left Side, 1 = Front, 2 = Right Side, 3 = Back, 4 = Top
                 //.....5 = Ground (underspill), 6 = DirectIntoSeeker, 7 = Miss
  string s_desc[8]; //.Create array of target surface names (ref above surfaces)
  int mainhitsurface; //.........................Surface that got hit the most
  //*****************************PARAMETERS*************************************
  //P_SEEKER----------------------------------------------------------------------
  double s_fov; //.............................deg (half angle of cone swept out)
  double s_aperture_diameter; //...............................................m
  double s_background; //...................background noise multiplier W/m^2/sr
  double s_signalnoise_threshold; //ratio of signal to noise necessary to detect
  double s_efficiency; //........efficiency factor accounting for various losses
  //P_DESIGNATOR------------------------------------------------------------------
  int d_lasertype; //.....................................0 = 1.06 um (Nd:Glass)
            //.....................................1 = 1.536 um (Er:Glass)
  int d_raycount; //...................Number of rays to divide laser pulse into
  double d_divergence; //.....................................................rad
  double d_pulse_energy; //....................................................J
  double d_pulse_frequency; //.................................................Hz
  double d_pulse_duration; //...................................................s
  double d_efficiency; //........efficiency factor accounting for various losses
  double d_h_error, d_v_error; //....designator error (hor and vert, rad, stdev)
  //P_TARGET----------------------------------------------------------------------
  Point3D t_size; //.....................................length, width, height: m
  double t_reflect[6]; // target reflectivity for each surface (ref surface des)
  //P_OTHER-----------------------------------------------------------------------
  int seed; //............................seed number for random number generator
  weather sal_weather; //..................struct containing weather information
  double s_noise; //...........................................................W
/*****************************************************************************/

private://************************---PRIVATE---*********************************

  //*********************************VARIABLES**********************************
  /*
  Rotations use RIGHT-HANDED coordinate system
  +z     +y
  |     /
  |    /
  |   /
  |  /
  | /
   - - - - - - - +x
  XYZ Zero Points:
  x = 0 - user-defined
  y = 0 - user-defined
  z = 0 - earth's surface
```

```
  Azimuth is angle between +x direction and
  projection of flight body onto x-y plane
  + Azimuth = CCW rotation as seen from above

  Elevation is angle between flight body and x-y plane
  + Elevation = Flight body above x-y plane

  */
  Point3D d_geo; //..............Designator location projected onto Earth-faced
                 //............. x/y/z coords (m) (See further description above)
  Point3D t_geo; //.............................Target location projected " " (m)
  Point3D s_geo; //.............................Target Location projected " " (m)
  Point2D s_orientation; //.................Azimuth/Elevation of projectile (deg)
                         //.......................(See further description above)
  double t_rotation; //..................deg (0 degrees is the target maneuvering
                     //.........................in the +y direction, + deg rotates
                     //.......................target clockwise as seen from above)
  //*********************************FUNCTIONS*************************************
  //F_MAIN---------------------------------------------------------------------
  void SetDefaults(); //...............Sets default values of initial parameters
  void LaserSpot //......Populates p vector with power, hit points, and surfaces
  (Point3D p1, Point3D p2, Point3D p3, Point2D s_or, double tr);
  //.................................................p1 = Designator Center (x,y,z)
  //..................................................p2 = Target Center (x,y,z)
  //..................................................p3 = Seeker Center (x,y,z)
  //...............................s_or = seeker orientation (azimuth, elevation)
  //.....................................................tr = Target Rotation
  void SignalLoss(); //...............Calculates signal loss through attenuation,
                     //...............reflection, and designator/receiver losses
  //F_UTILITY------------------------------------------------------------------
  // Calculates attenuation through atmosphere via lookup table
  double Attenuation(double currentsig,double traveldistance,double startheight,
                     double endheight, weather sal_weather, int lasertype);
  Point3D SpherCartConv(Point2D p1);// Converts azimuth/elevation to unit vector
  // Creates unit vector pointing from p1 to p2 (normalized by default)
  Point3D VectorPointing(Point3D p1, Point3D p2, bool norm = true);
  // Rotates vector by angle a (yaw) and b (pitch)
  Point3D VectorRotation(Point3D p1, double a, double b, int vect = 0);
  double VectorAngle(Point3D p1, Point3D p2); //.......Angle between two vectors
  double DotProduct(Point3D p1, Point3D p2); //........Dot product of two vectors
  void InverseMatrix(double m[3][3],double mi[3][3],double &det);//Inversematrix
  //*******************************CALCULATIONS************************************
  //C_TARGET_GEOMETRY----------------------------------------------------------
  vector <Point3D> s, n; //.........................s = Target surface coordinates
                         //............................n = Surface normal vectors
  vector <rayhit> r; //.....................Struct containing ray hit information
  //C_DIRECTION_VECTORS--------------------------------------------------------
  Point3D d_heading_vector; //.........................Designator pointing vector
  Point3D s_heading_vector; //............................Seeker pointing vector
  //C_ENERGY_LOSS--------------------------------------------------------------
  double s_aperture_area; //.....................................................m^2
  double s_t_multiplier; //..Ratio of power reaching seeker aperture from target
  //C_SEEKER_SIGNAL------------------------------------------------------------
  double s_power; //.............................................................W
  int s_hits[4]; //.................number of ray hits received in each quadrant
  quadsignal s_signal; //.................Total signal received in each quadrant
  quadsignal s_signalnoise; //..............Signal/Noise Ratio for each quadrant
  guidancesignal s_maneuver; //..Pitching and yawing maneuver guidance (-1 to 1)
  //C_DETECT-------------------------------------------------------------------
  bool s_encounter; //.....................True if target is within seeker's FOV
  bool s_detect; //................................True if laser S/N > threshold
};/*************************************************************************/
#endif
```

```
/*****************************************************************************/
```

s_sal_seeker_class.cpp

```
/*
SAL Seeker Class .cpp
Luke Strohm, Army Research Laboratory
3-14-11
*/

/*****************************************************************************/
#include "s_sal_seeker_class.h"
/*****************************************************************************/
sSalSeeker::sSalSeeker(){
  SetDefaults();
}/*****************************************************************************/
void sSalSeeker::SetDefaults(){
  //  Output Parameters
  printtoscreen = false;
  printtofile = false;
  //  Weather Parameters
  sal_weather.latitude = 1; //.....................0 = tropics (<23.5 deg +-)
                            //..............1 = mid-latitudes (23.5-50 deg +-)
                            //...................2 = sub-arctic (50-70 deg +-)
  sal_weather.season = 0; //............................0 = summer (3/22-9/21)
                          //............................1 = winter (9/22-3/21)
  sal_weather.visibility = 0; //............0 = clear (23 km), 1 = hazy (5 km)
  //  Seeker Parameters
  s_fov = 4.5; s_fov *= UTILITY.DEG2RAD; //...deg (half angle of cone swept out)
  s_aperture_diameter = 60E-3; //.....................m (based on 81mm mortar)
  s_background = 0.010; //.....................................W/m^2/sr
  s_signalnoise_threshold = 7.0;
  s_efficiency = 0.95; //.........efficiency factor encompassing various losses
  //  Designator (Scout) Parameters
  d_lasertype = 0; //...................................0 = 1.06 um (Nd:Glass)
            //.....................................1 = 1.536 um (Er:Glass)
  d_raycount = 10000; //.........................number of rays in laser beam
  d_divergence = 3E-4; //.....................................................rad
  d_pulse_energy = 80E-3; //...................................................J
  d_pulse_frequency = 10.0; //.................................................Hz
  d_pulse_duration = 15E-9; //.................................................s
  d_efficiency = 0.95; //.........efficiency factor encompassing various losses
  d_h_error = 1E-4; //................horizontal designator error (rad, stdev)
  d_v_error = 1E-4; //..................vertical designator error (rad, stdev)
  //  Target Parameters
  t_size.X = 6.4; //..........................................................m
  t_size.Y = 2.3; //..........................................................m
  t_size.Z = 2.3; //..........................................................m
  t_reflect[0] = 0.4; //.....................................................left side
  t_reflect[1] = 0.4; //.....................................................front
  t_reflect[2] = 0.4; //.....................................................right side
  t_reflect[3] = 0.4; //.....................................................back
  t_reflect[4] = 0.4; //.....................................................top
  t_reflect[5] = 0.4; //.....................................................ground
}/*****************************************************************************/
saloutputs sSalSeeker::S_SALSEEKER(Point3D d_geo, Point3D t_geo, Point3D s_geo,
                           Point2D s_orientation, double t_rotation, int seed){
  //  Set class variables equal to function inputs
  (*this).d_geo = d_geo;
  (*this).t_geo = t_geo;
  (*this).s_geo = s_geo;
  (*this).s_orientation = s_orientation;
  (*this).t_rotation = t_rotation;

  // Initialize random number generator
```

```
  STAT.MIRandGenInit(seed);

  //  Call Calculation Functions
  LaserSpot(d_geo, t_geo, s_geo, s_orientation, t_rotation);
  SignalLoss();

  programoutput.encounter = s_encounter;
  programoutput.detect = s_detect;
  programoutput.sal_signals = s_signal;
  programoutput.actuator_signals = s_maneuver;

  return programoutput;
}/*************************************************************************/

void sSalSeeker::LaserSpot
                (Point3D p1, Point3D p2, Point3D p3, Point2D s_or, double tr){
  //  Define local variables
  vector <Point3D> u(7), v(7); //..u and v (vectors from surface center to edge)
                               //..........SIDES: u (length dir), v (height dir)
                               //.....FRONT, BACK: u (width dir), v (height dir)
                               //.....TOP, BOTTOM: u (length dir), v (width dir)
  Point3D r_ind; //..............individual ray vector from designator to target
  double m[3][3], mi[3][3], dl[3], tuv[3];
  double det = 0.0; //.............................................determinant
  double r_mult = 0.0; //.......multiplier to get from designator to surface hit
  int hitcount = 0; //.................the number of hits on the surface most hit
  s_desc[0] = "Left Side"; s_desc[1] = "Front"; s_desc[2] = "Right Side";
  s_desc[3] = "Back"; s_desc[4] = "Top"; s_desc[5] = "Ground";
  s_desc[6] = "DirectToSeeker"; s_desc[7] = "Miss";

  //  Resize vectors s, n, and r
  s.resize(7); n.resize(7); r.resize(d_raycount);

  //  Initialize surface hit counts to 0
  surface[0] = surface[1] = surface[2] = surface[3] = surface[4]
             = surface[5] = surface[6] = surface[7] = 0;

  //  Target Rotation - only valid 0 to 2PI rad
  if ((tr < 0) || (tr > 2*M_PI)){
    cerr << "Invalid target angle" << endl;
    system("PAUSE");
    exit(1);
  }


  //  DEFINE TARGET SURFACE COORDINATES AND VECTORS*****************************

  //  Calculate Heading (Pointing) Vector for Seeker
  s_heading_vector = SpherCartConv(s_or);

  //  Target Surface Coordinates
  //  Side 0 (Left Side)
  s[0].X = p2.X - (t_size.Y/2) * cos(tr);
  s[0].Y = p2.Y + (t_size.Y/2) * sin(tr);
  s[0].Z = p2.Z;
  //  Side 1 (Front)
  s[1].X = p2.X + (t_size.X/2) * sin(tr);
  s[1].Y = p2.Y + (t_size.X/2) * cos(tr);
  s[1].Z = p2.Z;
  //  Side 2 (Right Side)
  s[2].X = p2.X + (t_size.Y/2) * cos(tr);
  s[2].Y = p2.Y - (t_size.Y/2) * sin(tr);
  s[2].Z = p2.Z;
```

```
//  Side 3 (Back)
s[3].X = p2.X - (t_size.X/2) * sin(tr);
s[3].Y = p2.Y - (t_size.X/2) * cos(tr);
s[3].Z = p2.Z;
//  Side 4 (Top)
s[4].X = p2.X;
s[4].Y = p2.Y;
s[4].Z = p2.Z + (t_size.Z/2);
//  Side 5 (Ground)
s[5].X = p2.X;
s[5].Y = p2.Y;
s[5].Z = 0;
//  Side 6 (DirectToSeeker)
s[6].X = p3.X;
s[6].Y = p3.Y;
s[6].Z = p3.Z;

//  Normal Vectors Pointing out of each surface
//  Side 0 (Left Side)
n[0].X = -cos(tr);
n[0].Y = sin(tr);
n[0].Z = 0;
//  Side 1 (Front)
n[1].X = sin(tr);
n[1].Y = cos(tr);
n[1].Z = 0;
//  Side 2 (Right Side)
n[2].X = cos(tr);
n[2].Y = -sin(tr);
n[2].Z = 0;
//  Side 3 (Back)
n[3].X = -sin(tr);
n[3].Y = -cos(tr);
n[3].Z = 0;
//  Side 4 (Top)
n[4].X = 0;
n[4].Y = 0;
n[4].Z = 1;
//  Side 5 (Ground)
n[5].X = 0;
n[5].Y = 0;
n[5].Z = 1;
//  Side 6 (DirectToSeeker)
n[6].X = s_heading_vector.X;
n[6].Y = s_heading_vector.Y;
n[6].Z = s_heading_vector.Z;

//  Surface-defining vectors (u and v)
//  Side 0 (Left Side)
u[0].X = n[1].X * t_size.X/2;
u[0].Y = n[1].Y * t_size.X/2;
u[0].Z = 0;
v[0].X = 0;
v[0].Y = 0;
v[0].Z = t_size.Z/2;
//  Side 1 (Front)
u[1].X = n[2].X * t_size.Y/2;
u[1].Y = n[2].Y * t_size.Y/2;
u[1].Z = 0;
v[1].X = 0;
v[1].Y = 0;
v[1].Z = t_size.Z/2;
//  Side 2 (Right Side)
```

```
u[2].X = n[3].X * t_size.X/2;
u[2].Y = n[3].Y * t_size.X/2;
u[2].Z = 0;
v[2].X = 0;
v[2].Y = 0;
v[2].Z = t_size.Z/2;
//  Side 3 (Back)
u[3].X = n[0].X * t_size.Y/2;
u[3].Y = n[0].Y * t_size.Y/2;
u[3].Z = 0;
v[3].X = 0;
v[3].Y = 0;
v[3].Z = t_size.Z/2;
//  Side 4 (Top)
u[4].X = n[1].X * t_size.X/2;
u[4].Y = n[1].Y * t_size.X/2;
u[4].Z = 0;
v[4].X = n[2].X * t_size.Y/2;
v[4].Y = n[2].Y * t_size.Y/2;
v[4].Z = 0;
//  Side 5 (Ground)
u[5].X = n[1].X * t_size.X/2;
u[5].Y = n[1].Y * t_size.X/2;
u[5].Z = 0;
v[5].X = n[2].X * t_size.Y/2;
v[5].Y = n[2].Y * t_size.Y/2;
v[5].Z = 0;
//  Side 6 (DirectToSeeker)
//  Define vectors that characterize seeker pointing
Point3D v1 = s_heading_vector;
Point3D v2 = VectorRotation(v1,0,0,1);
Point3D v3 = VectorRotation(v1,0,0,2);

u[6].X = v2.X * (s_aperture_diameter / 2);
u[6].Y = v2.Y * (s_aperture_diameter / 2);
u[6].Z = v2.Z * (s_aperture_diameter / 2);
v[6].X = v3.X * (s_aperture_diameter / 2);
v[6].Y = v3.Y * (s_aperture_diameter / 2);
v[6].Z = v3.Z * (s_aperture_diameter / 2);

//  DIVIDE LASER BEAM INTO INDIVIDUAL RAYS**********************************

//  Calculate Pointing Vector for Designator (with Error)
d_heading_vector = VectorRotation(VectorPointing(d_geo, t_geo),
                                  STAT.Normal(0,d_h_error),
                                  STAT.Normal(0,d_v_error));
for (int i=0;i<d_raycount;i++){

  /*  Normally distribute rays according to divergence
      Divergence for Gaussian beams assumed to be 2 st deviations
      Beam edge when intensity drops to I = Io/e^2    */

  r_ind = VectorRotation(d_heading_vector,
                         STAT.Normal(0,d_divergence/2),
                         STAT.Normal(0,d_divergence/2));

  //  Set default values
  r_mult = 1E10; //..............Any hits should produce r_mults under 1E10 m
  int tempsurface = 7;  //.......Surface 7 means that ray did not hit anything

  //  FIND WHAT SURFACE RAY HITS*********************************************
  for (int j=0;j<7;j++){
    //  Calculate m
```

```
  m[0][0] = -r_ind.X; m[0][1] = u[j].X; m[0][2] = v[j].X;
  m[1][0] = -r_ind.Y; m[1][1] = u[j].Y; m[1][2] = v[j].Y;
  m[2][0] = -r_ind.Z; m[2][1] = u[j].Z; m[2][2] = v[j].Z;
  //  Calculate mi and determinant
  InverseMatrix(m,mi,det);
  //  Calculate dl
  dl[0] = p1.X - s[j].X;
  dl[1] = p1.Y - s[j].Y;
  dl[2] = p1.Z - s[j].Z;
  //  Calculate TUV vector (T = multiplier, U = dim1, V = dim2)
  tuv[0] = mi[0][0]*dl[0] + mi[0][1]*dl[1] + mi[0][2]*dl[2];
  tuv[1] = mi[1][0]*dl[0] + mi[1][1]*dl[1] + mi[1][2]*dl[2];
  tuv[2] = mi[2][0]*dl[0] + mi[2][1]*dl[1] + mi[2][2]*dl[2];

  //  Determine if ray intersects plane
  if ((fabs(det)<1E-20) || tuv[0] <= 1E-20) continue;
  //  Determine if ray intersects surface on plane
  if (j != 5){ //.................................Ground plane is infinite
    //  Check intersection straight to seeker
    if (j == 6){
      double dist = sqrt(tuv[1]*tuv[1] + tuv[2]*tuv[2]);
      double fov_check = VectorAngle(s_heading_vector,
                                     VectorPointing(s_geo, d_geo));
      if ((dist > s_aperture_diameter/2) || (fov_check > s_fov)) continue;
    }
    //  Rectangular Target
    else if ((tuv[1] < -1) || (tuv[1] > 1) ||
             (tuv[2] < -1) || (tuv[2] > 1)) continue;
  }
  //  Look for lowest r_mult
  if (tuv[0] < r_mult){
    r_mult = tuv[0];
    tempsurface = j;
  }
}

//  RECORD HIT POINT AND SURFACE MOST HIT**********************************
//  Add ray hit point to array of hit points (p)
if (tempsurface == 6){ //.....................Ray goes directly into seeker
  r[i].hit.X = p3.X;
  r[i].hit.Y = p3.Y;
  r[i].hit.Z = p3.Z;
}
else if (tempsurface == 7){ //...............................Ray hits nothing
  r[i].hit.X = p1.X;
  r[i].hit.Y = p1.Y;
  r[i].hit.Z = p1.Z;
}
else{ //.......................................Ray reflecting off of surface
  r[i].hit.X = p1.X + r_ind.X * r_mult;
  r[i].hit.Y = p1.Y + r_ind.Y * r_mult;
  r[i].hit.Z = p1.Z + r_ind.Z * r_mult;
}

r[i].surf = tempsurface; //.......................Record surface hit by ray
surface[tempsurface]++; //........................Tally hits on each surface


//  Check ground hits to see if they are obscured from seeker by target
r[i].obscured = false;
if (tempsurface == 5)
{
  Point3D obs = VectorPointing(r[i].hit, s_geo);
```

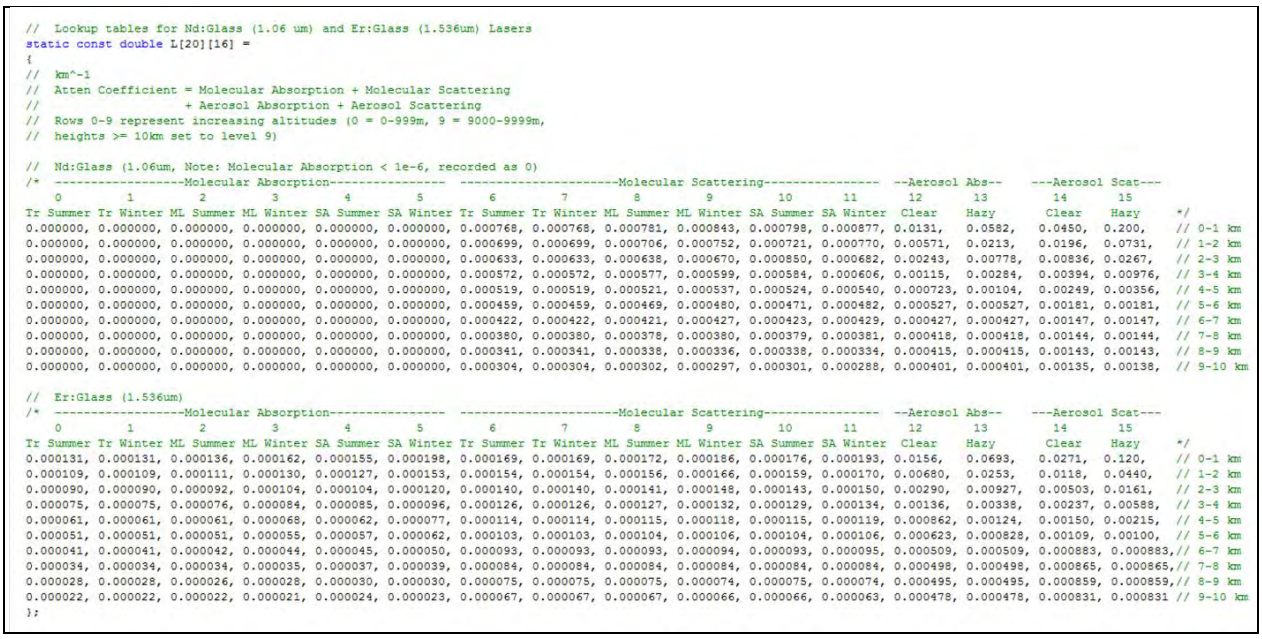
```
      double obsc_mult = 1E10;
      int obsc_surface = 5;

      for (int k=0;k<7;k++){
         //  Calculate m
        m[0][0] = -obs.X; m[0][1] = u[k].X; m[0][2] = v[k].X;
        m[1][0] = -obs.Y; m[1][1] = u[k].Y; m[1][2] = v[k].Y;
        m[2][0] = -obs.Z; m[2][1] = u[k].Z; m[2][2] = v[k].Z;
        //  Calculate mi and determinant
        InverseMatrix(m,mi,det);
        //  Calculate dl
        dl[0] = r[i].hit.X - s[k].X;
        dl[1] = r[i].hit.Y - s[k].Y;
        dl[2] = r[i].hit.Z - s[k].Z;
        //  Calculate TUV vector (T = multiplier, U = dim1, V = dim2)
        tuv[0] = mi[0][0]*dl[0] + mi[0][1]*dl[1] + mi[0][2]*dl[2];
        tuv[1] = mi[1][0]*dl[0] + mi[1][1]*dl[1] + mi[1][2]*dl[2];
        tuv[2] = mi[2][0]*dl[0] + mi[2][1]*dl[1] + mi[2][2]*dl[2];
        //  Determine if ray intersects plane
        if ((fabs(det)<1E-20) || tuv[0] <= 1E-20) continue;
        //  Determine if ray intersects surface on plane
        if ((tuv[1] < -1) || (tuv[1] > 1) ||
            (tuv[2] < -1) || (tuv[2] > 1)) continue;
        //  Look for lowest obsc_mult
        if (tuv[0] < obsc_mult){
          obsc_mult = tuv[0];
          obsc_surface = k; }
      }
      //  Determine if ray hit is obscured from seeker by target
      if (obsc_surface < 5) r[i].obscured = true;
    }
  }

  //  Determine "mainhitsurface"
  for (int l=0;l<8;l++)
  {
    if(surface[l] > hitcount)
    {
      hitcount = surface[l];
      mainhitsurface = l;
    }
  }

}/*************************************************************************/
void sSalSeeker::SignalLoss(){
  //  Initialize Quadrants
  s_signal.q1 = s_signal.q2 = s_signal.q3 = s_signal.q4 = 0;
  s_hits[0] = s_hits[1] = s_hits[2] = s_hits[3] = 0;
  //  Initialize S_Encounter and S_Detect to False (Default)
  s_encounter = s_detect = false;

  //  Define vectors
  Point3D d_t_vector; //.....................Distance between des and target (km)
  Point3D t_s_vector; //.......................Vector from target dot to seeker
  Point3D s_t_vector; //.......................Vector from seeker to target dot
  Point3D t_normal_vector; //....Vector pointing out of the illum target surface

  //  Define Distances/Rotations/Angles
  double d_t_distance; //...Straight-line distance between desig and target (km)
  double s_t_distance; //..Straight-line distance between seeker and target (km)
  double s_t_normal_angle; //......Angle betwn seeker pointing and target normal
  double s_t_correct_angle; //.............Angle that seeker is off from ray hit
  double r_correct_angle; //...................Individual FOV check for each ray
```

```
//  ENCOUNTER: Check to see if target is encountered by seeker
s_t_correct_angle = VectorAngle(s_heading_vector,VectorPointing(s_geo,t_geo));
if (s_t_correct_angle <= s_fov) s_encounter = true;

// Loop through rays and record the ones seeker receives
for (int i=0;i<d_raycount;i++){

  //  Initialize p variables to miss values
  r[i].power = 0.0;

  //  Throw out rays that miss or are obscured
  if (r[i].surf == 7 || r[i].obscured == true) continue;

  //*******************************GEOMETRY***********************************

  /*  Check if ray is within seeker's FOV (does not check reflection yet)
      The reason an extra check is done to see if the ray is encounter
      (vs. target) is because we may check instances where the designator
      greatly misses the target, and then the seeker sees the overspill/
      underspill, even though it is not looking at the target         */

  r_correct_angle = VectorAngle(s_heading_vector,
                                VectorPointing(s_geo,r[i].hit));
  if (r_correct_angle > s_fov) continue;

  //  Get Normal vector of surface
  t_normal_vector = n[r[i].surf];
  //  Calculate Distances
  d_t_distance = UTILITY.Distance3D(d_geo, r[i].hit); //......Desig-target (m)
  s_t_distance = UTILITY.Distance3D(s_geo, r[i].hit); //.....Seeker-target (m)
  //  Calculate Unit vectors Between Seeker, Desig, and Target Locations
  s_t_vector = VectorPointing(s_geo, r[i].hit);
  t_s_vector = VectorPointing(r[i].hit, s_geo);
  d_t_vector = VectorPointing(d_geo, r[i].hit);
  //  Calculate Angle between seeker and target surface normal
  s_t_normal_angle = VectorAngle(t_s_vector, t_normal_vector);
  //  Calculate Multiplier for Ratio of Reflected Radiation Received
  s_aperture_area = M_PI * pow(s_aperture_diameter / 2, 2);
  s_t_multiplier = (s_aperture_area/(M_PI * pow(s_t_distance,2)))
                * cos(s_t_normal_angle) * cos(r_correct_angle);
  //  Break out of loop if mult negative (surface normal > 90 degrees off)
  if (s_t_multiplier <= 1E-20) continue;
  //*************************POWER LOSS***********************************
  //  Divide up total signal into individual rays
  r[i].power = d_pulse_energy * d_efficiency /
               (d_raycount * d_pulse_duration);

  if (r[i].surf == 6){ //.............Ray goes directly from desig into seeker
    double d_s_distance = UTILITY.Distance3D(d_geo, s_geo);
    r[i].power = Attenuation(r[i].power, d_s_distance, d_geo.Z,
                             r[i].hit.Z, sal_weather, d_lasertype);
  }
  else{ //.........Ray reflects off of target or ground before reaching seeker
    //  Attenuate (Designator to Target)
    r[i].power = Attenuation(r[i].power, d_t_distance, d_geo.Z,
                             r[i].hit.Z, sal_weather, d_lasertype);
    //  Reflect off of Target
    r[i].power *= t_reflect[r[i].surf];
    //  Attenuate (Target to Seeker)
    r[i].power = Attenuation(r[i].power, s_t_distance, r[i].hit.Z,
                             s_geo.Z, sal_weather, d_lasertype);
    //  Decrease by distance and angle from target normal
```

```
  r[i].power *= s_t_multiplier;
}
r[i].power *= s_efficiency;
//*************DETERMINE SEEKER QUADRANT THAT RECEIVES RAY*****************
//  Define matrices to find where ray intersects surface
double m[3][3];
double mi[3][3];
double dl[3];
double tuv[3];
double det = 0.0;
//  Define vectors that characterize seeker pointing
Point3D v1 = s_heading_vector;
Point3D v2 = VectorRotation(s_heading_vector,0,0,1);
Point3D v3 = VectorRotation(s_heading_vector,0,0,2);
//  Calculate m
m[0][0] = v1.X; m[0][1] = v2.X; m[0][2] = v3.X;
m[1][0] = v1.Y; m[1][1] = v2.Y; m[1][2] = v3.Y;
m[2][0] = v1.Z; m[2][1] = v2.Z; m[2][2] = v3.Z;
//  Calculate mi and determinant
InverseMatrix(m,mi,det);
//  Calculate dl
if (r[i].surf == 6){
  dl[0] = d_geo.X - s_geo.X;
  dl[1] = d_geo.Y - s_geo.Y;
  dl[2] = d_geo.Z - s_geo.Z;
}
else{
  dl[0] = r[i].hit.X - s_geo.X;
  dl[1] = r[i].hit.Y - s_geo.Y;
  dl[2] = r[i].hit.Z - s_geo.Z;
}
//  Calculate TUV vector (T = multiplier, U = pitch, V = yaw)
tuv[0] = mi[0][0]*dl[0] + mi[0][1]*dl[1] + mi[0][2]*dl[2];
tuv[1] = mi[1][0]*dl[0] + mi[1][1]*dl[1] + mi[1][2]*dl[2];
tuv[2] = mi[2][0]*dl[0] + mi[2][1]*dl[1] + mi[2][2]*dl[2];
//  Determine what quadrant ray hits by pitch/yaw corrections
double length = tuv[0];
double yaw = tuv[1];
double pitch = tuv[2];

//  Sum up energies in each quadrant and divide by pulse width to get power
if (length > 0){
  if (pitch > 0){
    if (yaw > 0){ //.........................................+ Pitch, + Yaw
      s_hits[0]++;
      s_signal.q1 += r[i].power;
    }
    else{ //.................................................+ Pitch, - Yaw
      s_hits[1]++;
      s_signal.q2 += r[i].power;
    }
  }
  else{
    if (yaw > 0){ //.........................................- Pitch, + Yaw
      s_hits[2]++;
      s_signal.q3 += r[i].power;
    }
    else{ //.................................................- Pitch, - Yaw
      s_hits[3]++;
      s_signal.q4 += r[i].power;
    }
  }
}
```

```
}/*****************************************************************************/
  //*******************DETERMINE DETECT (S/N ABOVE THRESHOLD)*******************
  double q1 = s_signal.q1; double q2 = s_signal.q2;
  double q3 = s_signal.q3; double q4 = s_signal.q4;
  double s_solidangle = 2 * M_PI * (1 - cos(s_fov));

  //  Total seeker power is sum of quadrants
  s_power = q1 + q2 + q3 + q4;
  //  Calculate seeker noise
  s_noise = s_background * s_aperture_area * s_solidangle;
  //  Calculate each quadrant's S/N
  s_signalnoise.q1 = q1 / (s_noise / 4);
  s_signalnoise.q2 = q2 / (s_noise / 4);
  s_signalnoise.q3 = q3 / (s_noise / 4);
  s_signalnoise.q4 = q4 / (s_noise / 4);
  //  Detects if at least one quadrant's S/N > threshold
  if  (s_signalnoise.q1 > s_signalnoise_threshold ||
       s_signalnoise.q2 > s_signalnoise_threshold ||
       s_signalnoise.q3 > s_signalnoise_threshold ||
       s_signalnoise.q4 > s_signalnoise_threshold)
  s_detect = true;
  //************************DETERMINE MANEUVER SIGNALS**************************
  if (s_power < 1E-25){
    s_maneuver.pitch = s_maneuver.yaw = 0.0;
    s_maneuver.theta = s_maneuver.mag = 0.0;
  }
  else{
    s_maneuver.pitch = ((q1 + q2) - (q3 + q4)) / s_power;
    s_maneuver.yaw = ((q1 + q3) - (q2 + q4)) / s_power;

    double p = s_maneuver.pitch, y = s_maneuver.yaw;
    int quadcount = 0; double zerothresh = 1E-16;

    if (p >= 0) s_maneuver.theta = fabs(atan2(p,-y));
    else s_maneuver.theta = fabs(atan2(p,y)) + M_PI;
    if (q1 < zerothresh) quadcount++;
    if (q2 < zerothresh) quadcount++;
    if (q3 < zerothresh) quadcount++;
    if (q4 < zerothresh) quadcount++;
    if (quadcount > 1) s_maneuver.mag = 1.0;
    else s_maneuver.mag = sqrt((p*p + y*y) / 2);
  }
}/*****************************************************************************/
/*****************************************************************************/
/***************************** UTILITY FUNCTIONS *****************************/
/*****************************************************************************/
double sSalSeeker::Attenuation(double currentsig, double traveldistance,
                               double startheight, double endheight,
                               weather sal_weather, int lasertype){
  //  Weather
  int l = sal_weather.latitude; //.....................0 = tropics (< 30 deg lat)
                                //.............1 = mid-latitudes (30-60 deg lat)
                                //..................2 = sub-arctic (> 60 deg lat)
  int s = sal_weather.season; //.........................0 = summer (3/22-9/21)
```

```
                                //.........................1 = winter (9/22-3/21)
  int v = sal_weather.visibility; //..........0 = clear (23 km), 1 = hazy (5 km)
  int la = lasertype;             //.......................0 = Nd:Glass (1.06um)
                                  //.......................1 = Er:Glass (1.536um)

  //  Geometry and conversion to km
  double td = traveldistance * .001;
  double sh = startheight * .001;
  double eh = endheight * .001;
  double theta = asin((eh - sh) / td);

  //  Attenuation Coefficient and Attenuated Signal (Output of function)
  double att = 0;
  double attensig = currentsig;
```



```
// Lookup tables for Nd:Glass (1.06 um) and Er:Glass (1.536um) Lasers
static const double L[20][16] =
{
//  km^-1
//  Atten Coefficient = Molecular Absorption + Molecular Scattering
//                    + Aerosol Absorption + Aerosol Scattering
//  Rows 0-9 represent increasing altitudes (0 = 0-999m, 9 = 9000-9999m,
//  heights >= 10km set to level 9)

//  Nd:Glass (1.06um, Note: Molecular Absorption < 1e-6, recorded as 0)
/*  ------------------Molecular Absorption----------------  ----------------------Molecular Scattering----------------  --Aerosol Abs--    ---Aerosol Scat---
    0         1         2         3         4         5         6         7         8         9         10        11        12        13        14        15
Tr Summer Tr Winter ML Summer ML Winter SA Summer SA Winter Tr Summer Tr Winter ML Summer ML Winter SA Summer SA Winter  Clear     Hazy      Clear     Hazy     */
0.000000, 0.000000, 0.000000, 0.000000, 0.000000, 0.000000, 0.000768, 0.000768, 0.000781, 0.000843, 0.000798, 0.000877, 0.0131,   0.0582,   0.0450,  0.200,    // 0-1 km
0.000000, 0.000000, 0.000000, 0.000000, 0.000000, 0.000000, 0.000699, 0.000699, 0.000706, 0.000752, 0.000721, 0.000770, 0.00571,  0.0213,   0.0196,  0.0731,   // 1-2 km
0.000000, 0.000000, 0.000000, 0.000000, 0.000000, 0.000000, 0.000633, 0.000633, 0.000638, 0.000670, 0.000850, 0.000682, 0.00243,  0.00778,  0.00836, 0.0267,   // 2-3 km
0.000000, 0.000000, 0.000000, 0.000000, 0.000000, 0.000000, 0.000572, 0.000572, 0.000577, 0.000599, 0.000584, 0.000606, 0.00115,  0.00284,  0.00394, 0.00976,  // 3-4 km
0.000000, 0.000000, 0.000000, 0.000000, 0.000000, 0.000000, 0.000519, 0.000519, 0.000521, 0.000537, 0.000524, 0.000540, 0.000723, 0.00104,  0.00249, 0.00356,  // 4-5 km
0.000000, 0.000000, 0.000000, 0.000000, 0.000000, 0.000000, 0.000459, 0.000459, 0.000469, 0.000480, 0.000471, 0.000482, 0.000527, 0.000527, 0.00181, 0.00181,  // 5-6 km
0.000000, 0.000000, 0.000000, 0.000000, 0.000000, 0.000000, 0.000422, 0.000422, 0.000421, 0.000427, 0.000423, 0.000429, 0.000427, 0.000427, 0.00147, 0.00147,  // 6-7 km
0.000000, 0.000000, 0.000000, 0.000000, 0.000000, 0.000000, 0.000380, 0.000380, 0.000378, 0.000380, 0.000379, 0.000381, 0.000418, 0.000418, 0.00144, 0.00144,  // 7-8 km
0.000000, 0.000000, 0.000000, 0.000000, 0.000000, 0.000000, 0.000341, 0.000341, 0.000338, 0.000336, 0.000338, 0.000334, 0.000415, 0.000415, 0.00143, 0.00143,  // 8-9 km
0.000000, 0.000000, 0.000000, 0.000000, 0.000000, 0.000000, 0.000304, 0.000304, 0.000302, 0.000297, 0.000301, 0.000288, 0.000401, 0.000401, 0.00135, 0.00138,  // 9-10 km

//  Er:Glass (1.536um)
/*  ------------------Molecular Absorption----------------  ----------------------Molecular Scattering----------------  --Aerosol Abs--    ---Aerosol Scat---
    0         1         2         3         4         5         6         7         8         9         10        11        12        13        14        15
Tr Summer Tr Winter ML Summer ML Winter SA Summer SA Winter Tr Summer Tr Winter ML Summer ML Winter SA Summer SA Winter  Clear     Hazy      Clear     Hazy     */
0.000131, 0.000131, 0.000136, 0.000162, 0.000155, 0.000198, 0.000169, 0.000169, 0.000172, 0.000186, 0.000176, 0.000193, 0.0156,   0.0693,   0.0271,  0.120,    // 0-1 km
0.000109, 0.000109, 0.000111, 0.000130, 0.000127, 0.000153, 0.000154, 0.000154, 0.000156, 0.000166, 0.000159, 0.000170, 0.00680,  0.0253,   0.0118,  0.0440,   // 1-2 km
0.000090, 0.000090, 0.000092, 0.000104, 0.000104, 0.000120, 0.000140, 0.000140, 0.000141, 0.000148, 0.000143, 0.000150, 0.00290,  0.00927,  0.00503, 0.0161,   // 2-3 km
0.000075, 0.000075, 0.000076, 0.000084, 0.000085, 0.000096, 0.000126, 0.000126, 0.000127, 0.000132, 0.000129, 0.000134, 0.00136,  0.00338,  0.00237, 0.00588,  // 3-4 km
0.000061, 0.000061, 0.000061, 0.000068, 0.000062, 0.000077, 0.000114, 0.000114, 0.000115, 0.000118, 0.000115, 0.000119, 0.000862, 0.00124,  0.00150, 0.00215,  // 4-5 km
0.000051, 0.000051, 0.000051, 0.000055, 0.000057, 0.000062, 0.000103, 0.000103, 0.000104, 0.000106, 0.000104, 0.000106, 0.000623, 0.000828, 0.00109, 0.00100,  // 5-6 km
0.000041, 0.000041, 0.000042, 0.000044, 0.000045, 0.000050, 0.000093, 0.000093, 0.000093, 0.000094, 0.000093, 0.000095, 0.000509, 0.000509, 0.000883, 0.000883,// 6-7 km
0.000034, 0.000034, 0.000034, 0.000035, 0.000037, 0.000039, 0.000084, 0.000084, 0.000084, 0.000084, 0.000084, 0.000084, 0.000498, 0.000498, 0.000865, 0.000865,// 7-8 km
0.000028, 0.000028, 0.000026, 0.000028, 0.000030, 0.000030, 0.000075, 0.000075, 0.000075, 0.000074, 0.000075, 0.000074, 0.000495, 0.000495, 0.000859, 0.000859,// 8-9 km
0.000022, 0.000022, 0.000022, 0.000021, 0.000024, 0.000023, 0.000067, 0.000067, 0.000067, 0.000066, 0.000066, 0.000063, 0.000478, 0.000478, 0.000831, 0.000831 // 9-10 km
};
```

Figure B-1. Lookup tables for 1.06- and 1.536-µm lasers.

```
//  Lookup tables for Nd:Glass (1.06 um) and Er:Glass (1.536um) Lasers
static const double L[20][16] =
{
//  km^-1
//  Atten Coefficient = Molecular Absorption + Molecular Scattering
//                    + Aerosol Absorption + Aerosol Scattering
//  Rows 0-9 represent increasing altitudes (0 = 0-999m, 9 = 9000-9999m,
//  heights >= 10km set to level 9)

//  Nd:Glass (1.06um, Note: Molecular Absorption < 1e-6, recorded as 0)
/*  ------------------Molecular Absorption----------------  ----------------------
Molecular Scattering----------------  --Aerosol Abs--    ---Aerosol Scat---
      0         1         2         3         4         5         6         7
8         9         10        11        12        13        14        15
  Tr Summer Tr Winter ML Summer ML Winter SA Summer SA Winter Tr Summer Tr Winter ML
Summer ML Winter SA Summer SA Winter  Clear     Hazy      Clear     Hazy     */
  0.000000, 0.000000, 0.000000, 0.000000, 0.000000, 0.000000, 0.000768, 0.000768,
0.000781, 0.000843, 0.000798, 0.000877, 0.0131,   0.0582,   0.0450,  0.200,    // 0-1
km
  0.000000, 0.000000, 0.000000, 0.000000, 0.000000, 0.000000, 0.000699, 0.000699,
0.000706, 0.000752, 0.000721, 0.000770, 0.00571,  0.0213,   0.0196,  0.0731,   // 1-2
km
```

```
  0.000000, 0.000000, 0.000000, 0.000000, 0.000000, 0.000000, 0.000633, 0.000633,
0.000638, 0.000670, 0.000850, 0.000682, 0.00243,  0.00778,  0.00836, 0.0267,   // 2-3
km
  0.000000, 0.000000, 0.000000, 0.000000, 0.000000, 0.000000, 0.000572, 0.000572,
0.000577, 0.000599, 0.000584, 0.000606, 0.00115,  0.00284,  0.00394, 0.00976,  // 3-4
km
  0.000000, 0.000000, 0.000000, 0.000000, 0.000000, 0.000000, 0.000519, 0.000519,
0.000521, 0.000537, 0.000524, 0.000540, 0.000723, 0.00104,  0.00249, 0.00356,  // 4-5
km
  0.000000, 0.000000, 0.000000, 0.000000, 0.000000, 0.000000, 0.000459, 0.000459,
0.000469, 0.000480, 0.000471, 0.000482, 0.000527, 0.000527, 0.00181, 0.00181,  // 5-6
km
  0.000000, 0.000000, 0.000000, 0.000000, 0.000000, 0.000000, 0.000422, 0.000422,
0.000421, 0.000427, 0.000423, 0.000429, 0.000427, 0.000427, 0.00147, 0.00147,  // 6-7
km
  0.000000, 0.000000, 0.000000, 0.000000, 0.000000, 0.000000, 0.000380, 0.000380,
0.000378, 0.000380, 0.000379, 0.000381, 0.000418, 0.000418, 0.00144, 0.00144,  // 7-8
km
  0.000000, 0.000000, 0.000000, 0.000000, 0.000000, 0.000000, 0.000341, 0.000341,
0.000338, 0.000336, 0.000338, 0.000334, 0.000415, 0.000415, 0.00143, 0.00143,  // 8-9
km
  0.000000, 0.000000, 0.000000, 0.000000, 0.000000, 0.000000, 0.000304, 0.000304,
0.000302, 0.000297, 0.000301, 0.000288, 0.000401, 0.000401, 0.00135, 0.00138,  // 9-10
km

  //  Er:Glass (1.536um)
  /*  ------------------Molecular Absorption----------------  ----------------------
Molecular Scattering----------------  --Aerosol Abs--    ---Aerosol Scat---
      0         1         2         3         4         5         6         7
8         9         10        11        12        13        14        15
  Tr Summer Tr Winter ML Summer ML Winter SA Summer SA Winter Tr Summer Tr Winter ML
Summer ML Winter SA Summer SA Winter  Clear    Hazy       Clear    Hazy     */
  0.000131, 0.000131, 0.000136, 0.000162, 0.000155, 0.000198, 0.000169, 0.000169,
0.000172, 0.000186, 0.000176, 0.000193, 0.0156,   0.0693,   0.0271,  0.120,    // 0-1
km
  0.000109, 0.000109, 0.000111, 0.000130, 0.000127, 0.000153, 0.000154, 0.000154,
0.000156, 0.000166, 0.000159, 0.000170, 0.00680,  0.0253,   0.0118,  0.0440,   // 1-2
km
  0.000090, 0.000090, 0.000092, 0.000104, 0.000104, 0.000120, 0.000140, 0.000140,
0.000141, 0.000148, 0.000143, 0.000150, 0.00290,  0.00927,  0.00503, 0.0161,   // 2-3
km
  0.000075, 0.000075, 0.000076, 0.000084, 0.000085, 0.000096, 0.000126, 0.000126,
0.000127, 0.000132, 0.000129, 0.000134, 0.00136,  0.00338,  0.00237, 0.00588,  // 3-4
km
  0.000061, 0.000061, 0.000061, 0.000068, 0.000062, 0.000077, 0.000114, 0.000114,
0.000115, 0.000118, 0.000115, 0.000119, 0.000862, 0.00124,  0.00150, 0.00215,  // 4-5
km
  0.000051, 0.000051, 0.000051, 0.000055, 0.000057, 0.000062, 0.000103, 0.000103,
0.000104, 0.000106, 0.000104, 0.000106, 0.000623, 0.000828, 0.00109, 0.00100,  // 5-6
km
  0.000041, 0.000041, 0.000042, 0.000044, 0.000045, 0.000050, 0.000093, 0.000093,
0.000093, 0.000094, 0.000093, 0.000095, 0.000509, 0.000509, 0.000883, 0.000883,// 6-7
km
  0.000034, 0.000034, 0.000034, 0.000035, 0.000037, 0.000039, 0.000084, 0.000084,
0.000084, 0.000084, 0.000084, 0.000084, 0.000498, 0.000498, 0.000865, 0.000865,// 7-8
km
  0.000028, 0.000028, 0.000026, 0.000028, 0.000030, 0.000030, 0.000075, 0.000075,
0.000075, 0.000074, 0.000075, 0.000074, 0.000495, 0.000495, 0.000859, 0.000859,// 8-9
km
```

```
  0.000022, 0.000022, 0.000022, 0.000021, 0.000024, 0.000023, 0.000067, 0.000067,
0.000067, 0.000066, 0.000066, 0.000063, 0.000478, 0.000478, 0.000831, 0.000831 // 9-10
km
  };

  //  LEVEL PATH
  if (theta == 0){
    //  Figure out what height laser is in
    int row;
    int n = 0;
    do {
      row = n;
      n++;
    } while (sh >= n);
    if (row > 9) row = 9;

    //  Calculate attenuation coefficient for segment and attenuate the signal
    //      Molecular Absorption       Molecular Scattering
    att = L[row + 10*la][2*l + s] + L[row + 10*la][6 + 2*l + s]
    //      Aerosol Absorption          Aerosol Scattering
        + L[row + 10*la][12 + v]  + L[row + 10*la][14 + v];
    attensig *= exp(-att * td);
  }

  //  ANGLED PATH
  else{
    //  Initialize new variables
    double segment, ipart; double extrasegment = 0;
    int n, lowrow, highrow;

    //  Make sure start height is below end height
    if (theta < 0){
      double sh_temp = sh;
      double eh_temp = eh;
      sh = eh_temp;
      eh = sh_temp;
    }

    //  Find altitude row that signal starts in
    n = 0;
    do{
      lowrow = n;
      n++;
    } while (sh >= n);
    if (lowrow > 9) lowrow = 9;

    //  Find altitude row that signal ends in
    n = 0;
    do{
      highrow = n;
      n++;
    } while (eh >= n);

    if (highrow > 9){
      extrasegment = fabs((highrow - 9)/sin(theta));
      highrow = 9;
    }

    //  Loop through to attenuate signal through different altitudes
    for (int row = lowrow; row < highrow + 1; row++){
      //  Find out segment length
      if (lowrow == highrow) segment = td;
      else if (row == lowrow) segment = fabs((1 - modf(sh,&ipart))/sin(theta));
```

```
      else if (row == highrow)
      segment = fabs(modf(eh,&ipart)/sin(theta)) + extrasegment;
      else segment = fabs(1/sin(theta));

      //  Calculate attenuation coefficient for segment and attenuate the signal
      //      Molecular Absorption       Molecular Scattering
      att = L[row + 10*la][2*l + s] + L[row + 10*la][6 + 2*l + s]
      //      Aerosol Absorption          Aerosol Scattering
          + L[row + 10*la][12 + v]  + L[row + 10*la][14 + v];
      attensig *= exp(-att * segment);
    }
  }
  return attensig;
}/*************************************************************************/
Point3D sSalSeeker::VectorPointing(Point3D p1, Point3D p2, bool norm){
  //  Create vector from p1 to p2
  Point3D p3;
  double vec_mag = UTILITY.Distance3D(p1 , p2);
  //  Subtract to get vector (p3 = p2 - p1)
  p3.X = (p2.X-p1.X);
  p3.Y = (p2.Y-p1.Y);
  p3.Z = (p2.Z-p1.Z);
  //  Normalize vector if norm = true
  if (norm == true){
    p3.X /= vec_mag;
    p3.Y /= vec_mag;
    p3.Z /= vec_mag;
  }
  return p3;
}/*************************************************************************/
Point3D sSalSeeker::SpherCartConv(Point2D p1){
  //  Creates unit vector from azimuth/elevation angles
  Point3D p3;
  p3.X = cos(p1.Y) * cos(p1.X);
  p3.Y = cos(p1.Y) * sin(p1.X);
  p3.Z = sin(p1.Y);
  return p3;
}/*************************************************************************/
Point3D sSalSeeker::VectorRotation(Point3D p1, double a, double b, int vect){
  Point3D p3;
  //................Assume unit vectors for both original and transformed vector
  //.....................................p1 = original vector, a = yaw, b = pitch
  //.........................p3_p = Perturbed vector in new prime coord system
  double p3_p[3], p3_p_mag;

  switch(vect){
    case 0: // Arbitrary rotations (default case)
      p3_p[0] = 1; p3_p[1] = tan(a); p3_p[2] = tan(b); break;
    case 1: // 90 degree rotation to +y axis
      p3_p[0] = 0; p3_p[1] = 1; p3_p[2] = 0; break;
    case 2: // 90 degree rotation to +z axis
      p3_p[0] = 0; p3_p[1] = 0; p3_p[2] = 1; break;
  }

  //  Normalize p3_p[]
  p3_p_mag = sqrt(p3_p[0]*p3_p[0] + p3_p[1]*p3_p[1] + p3_p[2]*p3_p[2]);
  p3_p[0] *= 1/p3_p_mag; p3_p[1] *= 1/p3_p_mag; p3_p[2] *= 1/p3_p_mag;

  //  Create rotation matrix to rotate back to original
  //  coordinate system using two consecutive Euler rotations
  //  Rotation matrix derived by Robert Yager in ARL-TR-5520
  double R[3][3];
  double gamma = 1/(sqrt(1-p1.Z*p1.Z));
```

```
  R[0][0] = p1.X; R[0][1] = -gamma*p1.Y; R[0][2] = -gamma*p1.X*p1.Z;
  R[1][0] = p1.Y; R[1][1] =  gamma*p1.X; R[1][2] = -gamma*p1.Y*p1.Z;
  R[2][0] = p1.Z; R[2][1] =  0;          R[2][2] = 1/gamma;

  p3.X = R[0][0]*p3_p[0] + R[0][1]*p3_p[1] + R[0][2]*p3_p[2];
  p3.Y = R[1][0]*p3_p[0] + R[1][1]*p3_p[1] + R[1][2]*p3_p[2];
  p3.Z = R[2][0]*p3_p[0] + R[2][1]*p3_p[1] + R[2][2]*p3_p[2];

  return p3;
}/*************************************************************************/
double sSalSeeker::VectorAngle(Point3D p1, Point3D p2){
  //  Returns angle between two vectors
  double angle;

  //  Calculate vector magnitudes and dot product
  double p1_mag = sqrt(p1.X * p1.X + p1.Y * p1.Y + p1.Z * p1.Z);
  double p2_mag = sqrt(p2.X * p2.X +  p2.Y * p2.Y + p2.Z * p2.Z);
  double dotproduct = p1.X * p2.X + p1.Y * p2.Y + p1.Z * p2.Z;
  double division = dotproduct/(p1_mag*p2_mag);

  //  Check boundary conditions
  if (division > 1.0) division = 1.0;
  if (division < -1.0) division = -1.0;

  angle = acos(division);
  return angle;
}/*************************************************************************/
double sSalSeeker::DotProduct(Point3D p1, Point3D p2){
  return p1.X * p2.X + p1.Y * p2.Y + p1.Z * p2.Z;
}/*************************************************************************/
void sSalSeeker::InverseMatrix(double m[3][3], double mi[3][3], double &det){
  /*
  Calculates inverse matrix (mi) of matrix m
  Compute inverse matrix (3x3 specific)
                   -1                  T
  A = |a b c|     A  =    1   |A B C|       1   |A D G|
      |d e f|           ----- |D E F| =   ----- |B E H|
      |g h i|           det(A)|G H I|     det(A)|C F I|

  det = a(ei-fh) + b(fg-id) + c(dh-eg)
  */
  double a, b, c, d, e, f, g, h, i,
         A, B, C, D, E, F, G, H, I;

  a = m[0][0]; b = m[0][1]; c = m[0][2];
  d = m[1][0]; e = m[1][1]; f = m[1][2];
  g = m[2][0]; h = m[2][1]; i = m[2][2];

  A = e*i-f*h; B = f*g-d*i; C = d*h-e*g;
  D = c*h-b*i; E = a*i-c*g; F = b*g-a*h;
  G = b*f-c*e; H = c*d-a*f; I = a*e-b*d;

  det = a*(e*i-f*h) + b*(f*g-i*d) + c*(d*h-e*g);

  mi[0][0] = A/det; mi[0][1] = D/det; mi[0][2] = G/det;
  mi[1][0] = B/det; mi[1][1] = E/det; mi[1][2] = H/det;
  mi[2][0] = C/det; mi[2][1] = F/det; mi[2][2] = I/det;
}/*************************************************************************/
void sSalSeeker::PrintOutput(string filename, string format){
  string lat, season, vis, laser, q1, q2, q3, q4, summary;
  ofstream outfile;

  //  User specifies filename here if not specified in function call
```

```
if (printtofile == true && filename == ""){
  cout << "Please enter a filename: ";
  cin >> filename;
  cout << endl << endl;
  filename = "Output\\" + filename + format;
}
else filename += format;

//  Construct Quadsignal Percentages
if (s_signal.q1 == 0.0) q1 = "";
else q1 = ToString(s_signal.q1/s_power * 100,"%9.2f") + " %";
if (s_signal.q2 == 0.0) q2 = "";
else q2 = ToString(s_signal.q2/s_power * 100,"%9.2f") + " %";
if (s_signal.q3 == 0.0) q3 = "";
else q3 = ToString(s_signal.q3/s_power * 100,"%9.2f") + " %";
if (s_signal.q4 == 0.0) q4 = "";
else q4 = ToString(s_signal.q4/s_power * 100,"%9.2f") + " %";

//  Create description strings
switch(sal_weather.latitude){
  case 0: lat = "Tropical\n"; break;
  case 1: lat = "Mid-lat\n"; break;
  case 2: lat = "Sub-arctic\n"; break;
}
switch(sal_weather.season){
  case 0: season = "Summer\n"; break;
  case 1: season = "Winter\n"; break;
}
switch(sal_weather.visibility){
  case 0: vis = "Clear\n"; break;
  case 1: vis = "Hazy\n"; break;
}
switch(d_lasertype){
  case 0: laser = "Nd:Glass\n"; break;
  case 1: laser = "Er:Glass\n"; break;
}

//  Construct Summary String
string fill =  "                    ";
string fill2 = "              ";
summary = "\n" + Center("SAL SEEKER CLASS IMPLEMENTATION", ' ',"")
        + Center("developed for AWCB SWEEPM (LSS; 2011)", ' ',"") + "\n"
        + Center(" INPUT PARAMETERS ", '-',"")
        + "*** WEATHER\n"
        + "Latitude,                                              "
        + lat
        + "Season,                                                "
        + season
        + "Visibility,                                            "
        + vis + "\n"
        + "*** SEEKER\n"
        + "Seeker FOV (Half Angle),                               "
        + ToString(UTILITY.RAD2DEG * s_fov, "%-0.2f") + " degrees\n"
        + "Seeker Aperture Diameter,                              "
        + ToString(s_aperture_diameter * 1000, "%-0.2f") + " mm\n\n"
        + "*** DESIGNATOR\n"
        + "Laser Type,                                            "
        + laser
        + "Laser Divergence,                                      "
        + ToString(d_divergence * 1000, "%-0.2f") + " mrad\n"
        + "Ray Count,                                             "
        + ToString(d_raycount, "%-i") + "\n"
        + "Designator Error (H),                                  "
```

```
+ ToString(d_h_error * 1000, "%-0.2f") + " mrad\n"
+ "Designator Error (V),                                  "
+ ToString(d_v_error * 1000, "%-0.2f") + " mrad\n\n"
+ "*** TARGET\n"
+ "Size (Length; Width; Height),                         "
+ ToString(t_size.X, "%-0.1f") + ", "
+ ToString(t_size.Y, "%-0.1f") + ", "
+ ToString(t_size.Z, "%-0.1f") + "m\n"
+ "Reflectivity,\n"
+ "Left Side,                                            "
+ ToString(t_reflect[0], "%0.2f") + "\n"
+ "Front,                                                "
+ ToString(t_reflect[1], "%0.2f") + "\n"
+ "Right Side,                                           "
+ ToString(t_reflect[2], "%0.2f") + "\n"
+ "Back,                                                 "
+ ToString(t_reflect[3], "%0.2f") + "\n"
+ "Top,                                                  "
+ ToString(t_reflect[4], "%0.2f") + "\n"
+ "Ground,                                               "
+ ToString(t_reflect[5], "%0.2f") + "\n"
+ Center(" GEOMETRY ", '-',"")
+ "Designator to Target Distance,                        "
+ ToString(UTILITY.Distance3D(d_geo, t_geo)/1000, "%4.2f") + " km\n"
+ "Seeker to Target Distance,                            "
+ ToString(UTILITY.Distance3D(s_geo, t_geo)/1000, "%4.2f") + " km\n\n"
+ "Seeker Location (XYZ - m),                            "
+ ToString(s_geo.X, "%-0.0f") + ", "
+ ToString(s_geo.Y, "%-0.0f") + ", "
+ ToString(s_geo.Z, "%-0.0f") + "\n"
+ "Designator Location,                                  "
+ ToString(d_geo.X, "%-0.0f") + ", "
+ ToString(d_geo.Y, "%-0.0f") + ", "
+ ToString(d_geo.Z, "%-0.0f") + "\n"
+ "Target Location,                                      "
+ ToString(t_geo.X, "%-0.0f") + ", "
+ ToString(t_geo.Y, "%-0.0f") + ", "
+ ToString(t_geo.Z, "%-0.0f") + "\n\n"
+ "Seeker Heading,                                       <"
+ ToString(s_heading_vector.X, "%-0.2f") + ", "
+ ToString(s_heading_vector.Y, "%-0.2f") + ", "
+ ToString(s_heading_vector.Z, "%-0.2f") + ">\n"
+ "Designator Heading,                                   <"
+ ToString(d_heading_vector.X, "%-0.2f") + ", "
+ ToString(d_heading_vector.Y, "%-0.2f") + ", "
+ ToString(d_heading_vector.Z, "%-0.2f") + ">\n"
+ "Target Rotation,                                      "
+ ToString(UTILITY.RAD2DEG * t_rotation, "%-0.0f") + " degrees\n"
+ "Seeker Azimuth,                                       "
+ ToString(UTILITY.RAD2DEG * s_orientation.X, "%-0.0f") + " degrees\n"
+ "Seeker Elevation,                                     "
+ ToString(UTILITY.RAD2DEG * s_orientation.Y, "%-0.0f") + " degrees\n\n"
+ Center(" SURFACE HITS ", '-',"")
+ "Surface Most Hit,                                     "
+ s_desc[mainhitsurface] + "\n"
+ "Left Side,                                            "
+ ToString(surface[0], "%-5i") + ", ("
+ ToString((double)surface[0]*100 / d_raycount, "%-0.0f") + "%)\n"
+ "Front,                                                "
+ ToString(surface[1], "%-5i") + ", ("
+ ToString((double)surface[1]*100 / d_raycount, "%-0.0f") + "%)\n"
+ "Right Side,                                           "
+ ToString(surface[2], "%-5i") + ", ("
```

```
+ ToString((double)surface[2]*100 / d_raycount, "%-0.0f") + "%)\n"
+ "Back,                                                  "
+ ToString(surface[3], "%-5i") + ", ("
+ ToString((double)surface[3]*100 / d_raycount, "%-0.0f") + "%)\n"
+ "Top,                                                   "
+ ToString(surface[4], "%-5i") + ", ("
+ ToString((double)surface[4]*100 / d_raycount, "%-0.0f") + "%)\n"
+ "Ground,                                                "
+ ToString(surface[5], "%-5i") + ", ("
+ ToString((double)surface[5]*100 / d_raycount, "%-0.0f") + "%)\n"
+ "DirectIntoSeeker,                                      "
+ ToString(surface[6], "%-5i") + ", ("
+ ToString((double)surface[6]*100 / d_raycount, "%-0.0f") + "%)\n"
+ "Miss,                                                  "
+ ToString(surface[7], "%-5i") + ", ("
+ ToString((double)surface[7]*100 / d_raycount, "%-0.0f") + "%)\n\n"
+ Center(" SEEKER QUADRANT HITS ", '-',"")
+ "Q1 (+ Pitch; + Yaw),                                   "
+ ToString(s_hits[0], "%-i") + "\n"
+ "Q2 (+ Pitch; - Yaw),                                   "
+ ToString(s_hits[1], "%-i") + "\n"
+ "Q3 (- Pitch; + Yaw),                                   "
+ ToString(s_hits[2], "%-i") + "\n"
+ "Q4 (- Pitch; - Yaw),                                   "
+ ToString(s_hits[3], "%-i") + "\n\n"
+ Center(" POWER LOSS ", '-',"")
+ "Designator Laser Signal,                               "
+ ToString(d_pulse_energy / d_pulse_duration, "%-0.2e") + " W\n"
+ "Seeker Laser Signal,                                   "
+ ToString(s_power, "%-0.2e") + " W\n"
+ "Background Noise,                                      "
+ ToString(s_noise, "%-0.2e") + " W\n"
+ "S/N Threshold,                                         "
+ ToString(s_signalnoise_threshold, "%-0.2f") + "\n"
+ "S/N Q1,                                                "
+ ToString(s_signalnoise.q1, "%-0.2f") + "\n"
+ "S/N Q2,                                                "
+ ToString(s_signalnoise.q2, "%-0.2f") + "\n"
+ "S/N Q3,                                                "
+ ToString(s_signalnoise.q3, "%-0.2f") + "\n"
+ "S/N Q4,                                                "
+ ToString(s_signalnoise.q4, "%-0.2f") + "\n"
+ "S/N Total,                                             "
+ ToString(s_power / s_noise, "%-0.2f") + "\n\n"
+ Center(" ENCOUNTER/DETECT ", '-',"")
+ "0 = No; 1 = Yes\n"
+ "Target Encountered,                                    "
+ ToString(s_encounter, "%-i") + "\n"
+ "Target Detected,                                       "
+ ToString(s_detect, "%-i") + "\n\n"
+ Center(" QUADRANT SIGNALS ", '-',"") + "\n"
+ fill + Center("+ Pitch",' ',"",40)
+ fill + Center("",'-',"",40);
for (int n=0;n<3;n++) summary += fill + Center("|",' ',"|",40);
summary += fill + Align("Q1          |         Q2",'|',20,"|",40);
summary += fill + Center("|",' ',"|",40);
summary += fill + Align(q1 + "       |   " + q2,'|',20,"|",40);
for (int n=0;n<3;n++) summary += fill + Center("|",' ',"|",40);
summary += fill2
+ "+ Yaw |-------------------------------------| - Yaw\n";
for (int n=0;n<3;n++) summary += fill + Center("|",' ',"|",40);
summary += fill + Align("Q3          |         Q4",'|',20,"|",40);
summary += fill + Center("|",' ',"|",40);
```

```
            summary += fill + Align(q3 + "      |  " + q4,'|',20,"|",40);
            for (int n=0;n<3;n++) summary += fill + Center("|",' ',"|",40);
            summary += fill + Center("",'-',"",40)
            + fill + Center("- Pitch",' ',"",40)
            + "\n\n"
            + Center(" MANEUVER GUIDANCE ", '-',"") + "\n"
            + "Pitch Command,                                          "
            + ToString(s_maneuver.pitch, "%-0.2f") + "\n"
            + "Yaw Command,                                            "
            + ToString(s_maneuver.yaw, "%-0.2f") + "\n"
            + "Signal Center: Theta,                                   "
            + ToString(UTILITY.RAD2DEG * s_maneuver.theta, "%-0.2f") + "\n"
            + "Signal Center: Magnitude,                               "
            + ToString(s_maneuver.mag, "%-0.2f") + "\n\n"
            + Center("", '-',"") + "\n"
            ;

  if (printtoscreen == true) cout << summary << endl;
  if (printtofile == true){
    outfile.open(filename.c_str());
    outfile << summary << endl;
    outfile.close();
  }
}/*****************************************************************************/
```

main.cpp #1 (Embeddable)

```
/*
SAL Seeker Class Basic Implementation (main.cpp)
Luke Strohm, Army Research Laboratory
3-14-11
*/

/*****************************************************************************/
#include <s_sal_seeker_class.h>
/*****************************************************************************/
int main()
{
  sSalSeeker RUN;

  //********************DEFINE INPUT/OUTPUT VARIABLES************************//
  //  INPUT-------------------------------------------------------------------
  Point3D d_geo, t_geo, s_geo; //.....................Designator, Target, Seeker
  Point2D s_orientation; //.........................Azimuth, Elevation (degrees)
  double t_rotation; //.........degrees (0 is target traveling in the +y direct)
                     //..........rotation is clockwise as seen from above target
  int seed; //...................................Seed for random number generator
  //  OUTPUT------------------------------------------------------------------
  // Mega-struct contains encounter, detect, quad signals, and maneuver guidance
  saloutputs results;

  //*******************************SET INPUTS********************************//
    s_geo.X = -5000; s_geo.Y = 0; s_geo.Z = 1.15;
    d_geo.X = -2000; d_geo.Y = 0; d_geo.Z = 1.15;
    t_geo.X = 0    ; t_geo.Y = 0; t_geo.Z = 1.15;
    s_orientation.X = 0; s_orientation.Y = 0; t_rotation = 0;
    seed = 0;

  //*******************************RUN PROGRAM*******************************//
  //  Convert inputs (degrees to radians)
  s_orientation.X *= RUN.UTILITY.DEG2RAD;
  s_orientation.Y *= RUN.UTILITY.DEG2RAD;
```

```
  t_rotation *= RUN.UTILITY.DEG2RAD;

  //  Run main calculation function
  results = RUN.S_SALSEEKER(d_geo,t_geo,s_geo,s_orientation,t_rotation,seed);

  RUN.printtoscreen = true;
  RUN.printtofile = false;
  RUN.PrintOutput("", ".txt");

  system("PAUSE");
  return 0;
}
```

main.cpp #2 (Stand-Alone)

```
/*
SAL Seeker Class implementation (main.cpp)
Luke Strohm, Army Research Laboratory
3-14-11
*/

/*****************************************************************************/
#include <s_sal_seeker_class.h>
/*****************************************************************************/
int main()
{
  sSalSeeker RUN;
/*****************************************************************************/
  //  Run Options
  bool batchrun = false;
  bool testinputs = false;
  RUN.printtoscreen = true;
  RUN.printtofile = false;
  string oformat = ".txt";

  /************************DEFINE INPUT/OUTPUT VARIABLES**********************/
  //INPUT-----------------------------------------------------------------------
  Point3D d_geo, t_geo, s_geo; //.....................Designator, Target, Seeker
  Point2D s_orientation; //..........................Azimuth, Elevation (degrees)
  double t_rotation; //.........degrees (0 is target traveling in the +y direct)
                     //..........rotation is clockwise as seen from above target
  int seed; //................................Seed for random number generator

  //OUTPUT----------------------------------------------------------------------
  // Mega-struct contains encounter, detect, quad signals, and maneuver guidance
  saloutputs results;

  /*************************RUN PROGRAM(SINGLE/BATCH)************************/
  //  Initialize list of file names and find first txt file in folder
  _finddata_t filenamelist; //......._finddata_t is a Windows-specific structure
  int hfile; //.....................................................first file
  string ifile, ofile, ofile_summary, outputstring; //.................i/o files
  ofile_summary = "Output\\OutputSummary.csv";

  //  Set input folder and output summary file
  if (batchrun == true) hfile = _findfirst("Input\\*.txt", &filenamelist);
  else hfile = _findfirst("*.txt", &filenamelist);

  //  Open output summary file
  ofstream FileToWrite;
  FileToWrite.open(ofile_summary.c_str());
  int counter = 1;

  do{
    //SET UP I/O FILES------------------------------------------------------------
```

```
//  Define input and output folder locations
if (batchrun == true){
  ifile = "Input\\";
  ifile = ifile + filenamelist.name;
}
else ifile = filenamelist.name;
//  Open next input file in the folder
ifstream FileToRead;
FileToRead.open(ifile.c_str());
// If can't open input file, terminate with error
if (!FileToRead){
  cout << "Unable to open file\n";
  system("PAUSE");
  exit(1);
}

/*************************GET DATA FROM INPUT FILE************************/
//  Inputs are delimited by (;)
for (int i=0;i<44;i++){
  getline(FileToRead, outputstring, ';');
  if (outputstring == ""){
    getline(FileToRead, outputstring);
    continue;
  }
  switch(i){
    case 1: s_geo.X = atof(outputstring.c_str()); break;
    case 2: s_geo.Y = atof(outputstring.c_str()); break;
    case 3: s_geo.Z = atof(outputstring.c_str()); break;
    case 4: d_geo.X = atof(outputstring.c_str()); break;
    case 5: d_geo.Y = atof(outputstring.c_str()); break;
    case 6: d_geo.Z = atof(outputstring.c_str()); break;
    case 7: t_geo.X = atof(outputstring.c_str()); break;
    case 8: t_geo.Y = atof(outputstring.c_str()); break;
    case 9: t_geo.Z = atof(outputstring.c_str()); break;
    case 10: t_rotation = atof(outputstring.c_str()); break;
    case 11: s_orientation.X = atof(outputstring.c_str()); break;
    case 12: s_orientation.Y = atof(outputstring.c_str()); break;
    case 13: seed = atoi(outputstring.c_str()); break;
    case 14: break; //.................................input file line break
    case 15: RUN.sal_weather.latitude = atoi(outputstring.c_str()); break;
    case 16: RUN.sal_weather.season = atoi(outputstring.c_str()); break;
    case 17: RUN.sal_weather.visibility = atoi(outputstring.c_str()); break;
    case 18: break; //.................................input file line break
    case 19: RUN.s_fov = atof(outputstring.c_str()) * RUN.UTILITY.DEG2RAD;
             break;
    case 20: RUN.s_aperture_diameter = atof(outputstring.c_str()); break;
    case 21: RUN.s_background = atof(outputstring.c_str()); break;
    case 22: RUN.s_signalnoise_threshold = atof(outputstring.c_str());break;
    case 23: RUN.s_efficiency = atof(outputstring.c_str()); break;
    case 24: break; //.................................input file line break
    case 25: RUN.d_lasertype = atoi(outputstring.c_str()); break;
    case 26: RUN.d_raycount = atoi(outputstring.c_str()); break;
    case 27: RUN.d_divergence = atof(outputstring.c_str()); break;
    case 28: RUN.d_pulse_energy = atof(outputstring.c_str()); break;
    case 29: RUN.d_pulse_frequency = atof(outputstring.c_str()); break;
    case 30: RUN.d_pulse_duration = atof(outputstring.c_str()); break;
    case 31: RUN.d_efficiency = atof(outputstring.c_str()); break;
    case 32: RUN.d_h_error = atof(outputstring.c_str()); break;
    case 33: RUN.d_v_error = atof(outputstring.c_str()); break;
    case 34: break; //.................................input file line break
    case 35: RUN.t_size.X = atof(outputstring.c_str()); break;
    case 36: RUN.t_size.Y = atof(outputstring.c_str()); break;
    case 37: RUN.t_size.Z = atof(outputstring.c_str()); break;
```

```
    case 38: RUN.t_reflect[0] = atof(outputstring.c_str()); break;
    case 39: RUN.t_reflect[1] = atof(outputstring.c_str()); break;
    case 40: RUN.t_reflect[2] = atof(outputstring.c_str()); break;
    case 41: RUN.t_reflect[3] = atof(outputstring.c_str()); break;
    case 42: RUN.t_reflect[4] = atof(outputstring.c_str()); break;
    case 43: RUN.t_reflect[5] = atof(outputstring.c_str()); break;
  }
  getline(FileToRead, outputstring);
}

/**********************DISPLAY UPDATED INPUTS ON SCREEN********************/
if (testinputs == true){
     cout << "S_GEO_X = " << s_geo.X << endl
       << "S_GEO_Y = " << s_geo.Y << endl
       << "S_GEO_Z = " << s_geo.Z << endl
       << "D_GEO_X = " << d_geo.X << endl
       << "D_GEO_Y = " << d_geo.Y << endl
       << "D_GEO_Z = " << d_geo.Z << endl
       << "T_GEO_X = " << t_geo.X << endl
       << "T_GEO_Y = " << t_geo.Y << endl
       << "T_GEO_Z = " << t_geo.Z << endl
       << "T_Rotation = " << t_rotation << endl
       << "Azimuth = " << s_orientation.X << endl
       << "Elevation = " << s_orientation.Y << endl
       << "Random # Seed = " << seed << endl
       << "Latitude = " << RUN.sal_weather.latitude << endl
       << "Season = " << RUN.sal_weather.season << endl
       << "Visibility = " << RUN.sal_weather.visibility << endl
       << "FOV = " << RUN.s_fov * RUN.UTILITY.RAD2DEG << endl
       << "Aperture = " << RUN.s_aperture_diameter << endl
       << "Seeker Background Noise Multiplier = " << RUN.s_background << endl
       << "S/N Threshold = " << RUN.s_signalnoise_threshold << endl
       << "Seeker Efficiency = " << RUN.s_efficiency << endl
       << "Laser Type = " << RUN.d_lasertype << endl
       << "Ray Count = " << RUN.d_raycount << endl
       << "Divergence = " << RUN.d_divergence << endl
       << "Pulsed Energy = " << RUN.d_pulse_energy << endl
       << "Pulse Frequency = " << RUN.d_pulse_frequency << endl
       << "Pulse Duration = " << RUN.d_pulse_duration << endl
       << "Designator Efficiency = " << RUN.d_efficiency << endl
       << "D_H Error = " << RUN.d_h_error << endl
       << "D_V Error = " << RUN.d_v_error << endl
       << "TSize X = " << RUN.t_size.X << endl
       << "TSize Y = " << RUN.t_size.Y << endl
       << "TSize Z = " << RUN.t_size.Z << endl
       << "TReflect[0] = " << RUN.t_reflect[0] << endl
       << "TReflect[1] = " << RUN.t_reflect[1] << endl
       << "TReflect[2] = " << RUN.t_reflect[2] << endl
       << "TReflect[3] = " << RUN.t_reflect[3] << endl
       << "TReflect[4] = " << RUN.t_reflect[4] << endl
       << "TReflect[5] = " << RUN.t_reflect[5] << endl << endl;
  system("PAUSE");
}
//  Close input file
FileToRead.close();

//CONVERT FROM DEGREES TO RADIANS-----------------------------------------
s_orientation.X *= RUN.UTILITY.DEG2RAD;
s_orientation.Y *= RUN.UTILITY.DEG2RAD;
t_rotation *= RUN.UTILITY.DEG2RAD;

/*****************************CALCULATIONS*******************************/
//  Run main calculation function
```

```
results = RUN.S_SALSEEKER(d_geo,t_geo,s_geo,s_orientation,t_rotation,seed);

/*********************************OUTPUT*********************************/

//  Create output file name
ofile = "Output\\Seek(" + ToString(s_geo.X, "%-0.0f") + "_"
      + ToString(s_geo.Y, "%-0.0f") + "_"
      + ToString(s_geo.Z, "%-0.0f") + ")"
      + "Desig(" + ToString(d_geo.X, "%-0.0f") + "_"
      + ToString(d_geo.Y, "%-0.0f") + "_"
      + ToString(d_geo.Z, "%-0.0f") + ")"
      + "Az" + ToString(s_orientation.X * RUN.UTILITY.RAD2DEG, "%-0.0f")
      + "_"
      + "El" + ToString(s_orientation.Y * RUN.UTILITY.RAD2DEG, "%-0.0f")
      + "_"
      + "TR" + ToString(t_rotation * RUN.UTILITY.RAD2DEG, "%-0.0f");

//  Output File Options
if (RUN.printtoscreen == true || RUN.printtofile == true)
  RUN.PrintOutput(ofile, oformat);

//  Write output to general summary file
double totalsig = results.sal_signals.q1
                + results.sal_signals.q2
                + results.sal_signals.q3
                + results.sal_signals.q4;
double totalSN = totalsig/RUN.s_noise;
if (counter == 1){
  FileToWrite << "OUTPUT SUMMARY"
              << endl << endl << endl
              << "-------Seeker-------,,,-----Designator-----,,,,"
              << "-------Hit Locations-------"
              << endl
              << "X,Y,Z,X,Y,Z,Az,El,TR,DHe,DVe,MAIN HIT,"
              << "FS,F,BS,B,T,G,DirS,Miss,Enc,Det,"
              << "Q1,Q2,Q3,Q4,TotSig,TotS/N,Pitch,Yaw,Theta,r"
              << endl;
}
FileToWrite << s_geo.X << "," << s_geo.Y << "," << s_geo.Z << ","
            << d_geo.X << "," << d_geo.Y << "," << d_geo.Z << ","
            << s_orientation.X * RUN.UTILITY.RAD2DEG << ","
            << s_orientation.Y * RUN.UTILITY.RAD2DEG << ","
            << t_rotation * RUN.UTILITY.RAD2DEG << ","
            << RUN.d_h_error * 1000 << ","
            << RUN.d_v_error * 1000 << ","
            << RUN.s_desc[RUN.mainhitsurface] << ","
            << RUN.surface[0] << "," << RUN.surface[1] << ","
            << RUN.surface[2] << "," << RUN.surface[3] << ","
            << RUN.surface[4] << "," << RUN.surface[5] << ","
            << RUN.surface[6] << "," << RUN.surface[7] << ","
            << results.encounter << ","
            << results.detect << ","
            << results.sal_signals.q1 * 1000 << ","
            << results.sal_signals.q2 * 1000 << ","
            << results.sal_signals.q3 * 1000 << ","
            << results.sal_signals.q4 * 1000 << ","
            << totalsig * 1000 << ","
            << totalSN << ","
            << results.actuator_signals.pitch << ","
            << results.actuator_signals.yaw << ","
            << results.actuator_signals.theta * RUN.UTILITY.RAD2DEG << ","
            << results.actuator_signals.mag << ","
            << endl;
```

```
    counter++;
 }
 while (_findnext(hfile,&filenamelist) == 0); //.......Loop through input files

//*************************************************************************//
  //  Close I/O files
  _findclose(hfile); //.................................Close the search handle
  FileToWrite.close(); //...............................Close run summary file
//*************************************************************************//
  if (RUN.printtoscreen == true) system("PAUSE");
  return 0;
}//*************************************************************************//
```

INTENTIONALLY LEFT BLANK.

Appendix C. Utilities Code

Utilities Class

Written by Mary Arthur of the U.S. Army Research Laboratory.

Structs: Point2D, Point3D

Functions: Distance3D, DEG2RAD, RAD2DEG

Statistics Class

Written by Mary Arthur of the U.S. Army Research Laboratory. An implementation of the routines found in

Saucier, R. *Computer Generation of Statistical Distributions*, ARL-TR-2168. U.S. Army Research Laboratory: Aberdeen Proving Ground, MD, March 2000.

Functions:

- Normal – returns a random draw based on a normal distribution

Y_Format Namespace

Written by Robert Yager of the U.S. Army Research Laboratory. Y_format makes it easy to create an elegant output display, and it was used in the "PrintOutput" function of *sSalSeeker*.

y_format_namespace.h

```
   /*************************************************************************
 *************************************************************************
***                                                                     ***
```

```
**                          STRING FORMATTING FUNCTIONS                          **
**                           version 1.00 (01-23-2011)                           **
**                                  -Rob Yager                                   **
***                                                                             ***
 ********************************************************************************
    ********************************************************************************/
#ifndef Y_FORMAT_NAMESPACE_H_
#define Y_FORMAT_NAMESPACE_H_
/*******************************************************************************/
#include <string>
using std::string;
/*******************************************************************************/
namespace yFormat{
  string Center(string s,char fill=' ',string border=" #",int width=80);
  string Align(string s,char center,int n,string border=" #",int width=80);
  string Align2(string s,char center,int n,string s2,int n2,string border=" #");
  string ToString(double number,string format="%f");
  string ToString(int number,string format="%d");
  string ToString(string text,string format="%s");
};/*******************************************************************************/
#endif/*******************************************************************************/
```

y_format_namespace.cc

```
#define _CRT_SECURE_NO_WARNINGS//.................disables deprecation warnings
#include "y_format_namespace.h"
/*******************************************************************************/
string yFormat::Center(string s,char c,string border,int width){
  //reverse the order of characters in border and copy to rborder
  string rborder;
  for(int i=0;i<(int)border.size();++i){
    rborder+=border[border.size()-i-1];
  }
  //calculate the amount of white space
  double white_space=(width-s.size()-2*border.size())/2.0;
  //create the centered string
  string out;
  out+=border;
  for(int i=0;i<(int)white_space;++i) out+=c;
  out+=s;
  for(int i=0;i<(int)(white_space+0.501);++i) out+=c;
  out+=rborder+"\n";
  return out;
}/*******************************************************************************/
string yFormat::Align(string s,char c,int n,string border,int width){
  int count=0;
  for(count=0;count<(int)s.size()&&s[count]!=c;++count);// if(s[i]==c) count=i;
  string out=border;
  for(int i=0;i<n-count-(int)border.size()-1;++i) out+=' ';
  out+=s;
  int whitespace=width-(int)out.size()-(int)border.size();
  for(int i=0;i<whitespace;++i) out+=' ';
  //reverse the order of characters in border and copy to rborder
  string rborder;
  for(int i=0;i<(int)border.size();++i){
    rborder+=border[border.size()-i-1];
  }
  out+=rborder+"\n";
  return out;
}/*******************************************************************************/
string yFormat::Align2(string s,char c,int n,string s2,int n2,string border){
  int count=0;
  for(count=0;count<(int)s.size()&&s[count]!=c;++count);// if(s[i]==c) count=i;
  string out=border;
  for(int i=0;i<n-count-(int)border.size()-1;++i) out+=' ';
```

```
  out+=s;

  int whitespace=n2-(int)out.size();
  for(int i=0;i<whitespace;++i) out+=' ';
  out+=s2;
  whitespace=80-(int)out.size()-(int)border.size();

  //reverse the order of characters in border and copy to rborder
  string rborder;
  for(int i=0;i<(int)border.size();++i){
    rborder+=border[border.size()-i-1];
  }
  out+=rborder+"\n";
  return out;
}/*************************************************************************/
string yFormat::ToString(double number,string format){
  char out[50];
  sprintf(out,format.c_str(),number);
  string string_out=out;
  return string_out;
}/*************************************************************************/
string yFormat::ToString(int number,string format){
  char out[50];
  sprintf(out,format.c_str(),number);
  string string_out=out;
  return string_out;
}/*************************************************************************/
string yFormat::ToString(string text,string format){
  char out[200];
  sprintf(out,format.c_str(),text.c_str());
  string string_out=out;
  return string_out;
}/*************************************************************************/
```

NO. OF COPIES	ORGANIZATION
1 (PDF only)	DEFENSE TECHNICAL INFORMATION CTR DTIC OCA 8725 JOHN J KINGMAN RD STE 0944 FORT BELVOIR VA 22060-6218
1	DIRECTOR US ARMY RESEARCH LAB IMNE ALC HRR 2800 POWDER MILL RD ADELPHI MD 20783-1197
1	DIRECTOR US ARMY RESEARCH LAB RDRL CIO LL 2800 POWDER MILL RD ADELPHI MD 20783-1197
1	DIRECTOR US ARMY RESEARCH LAB RDRL CIO MT 2800 POWDER MILL RD ADELPHI MD 20783-1197
1	DIRECTOR US ARMY RESEARCH LAB RDRL D 2800 POWDER MILL RD ADELPHI MD 20783-1197

NO. OF
COPIES ORGANIZATION

ABERDEEN PROVING GROUND

37 DIR USARL
(35 HC RDRL WM
2 CD) P PLOSTINS
M ZOLTOSKI
RDRL WML
J NEWILL
E SCHMIDT
T VONG
P WEINACHT
RDRL WML A
M ARTHUR
B BREECH
C MUNSON
W OBERLE (CD ONLY)
C PATTERSON
R PEARSON
L STROHM (1 HC, 1 CD)
A THOMPSON
P WYANT
R YAGER
RDRL WML B
J MORRIS
RDRL WML C
B ROOS
RDRL WML D
R BEYER
RDRL WML E
I CELMINS
F FRESCONI
S SILTON
RDRL WML F
R HALL
D HEPNER
K HUBBARD
P HUFNAL
M ILG
G KATULKA
D LYON
D PETRICK
B TOPPER
RDRL WML G
T G BROWN
RDRL WML H
B SORENSEN
RDRL WMM
J BEATTY
RDRL WMP
P BAKER
RDRL WMS
T ROSENBERGER

INTENTIONALLY LEFT BLANK.